Secret-Shared Shuffle from Authenticated Correlations

Xiangfu Song[1], Xiaojian Liang[2], Ye Dong[3(✉)], Jianli Bai[4], Pu Duan[2], Changyu Dong[5], Tianwei Zhang[1], and Ee-Chien Chang[3]

[1] Nanyang Technological University, Singapore, Singapore
{xiangfu.song,tianwei.zhang}@ntu.edu.sg
[2] Ant International, Ant Group, Hangzhou, China
p.duan@antgroup.com
[3] National University of Singapore, Singapore, Singapore
dongye@nus.edu.sg, changec@comp.nus.edu.sg
[4] Singapore Management University, Singapore, Singapore
[5] Guangzhou University, Guangzhou, China
changyu.dong@gzhu.edu.cn

Abstract. Shuffle is a basic primitive for secure computation. Secret-shared shuffle refers to oblivious permutation over secret-shared data, which has broad applications in secret-sharing-based secure computation. Since shuffle is typically used in highly sensitive applications, malicious security is often necessary to provide realistic security guarantees. This paper proposes a new family of two-party maliciously secure secret-shared shuffle protocols with linear communication/computation cost and constant-round communication. Achieving this goal has been proven non-trivial by several recent attempts. We answer this question by proposing a new and simple shuffle paradigm based on authenticated correlations. We start by proposing a simple and efficient protocol template based on authenticated correlations with linear cost and constant-round communication. The protocol can be enhanced to be fully authenticated against a malicious sender, which avoids selective-failure attacks that incur the main overhead in existing solutions. However, our roadmap introduces a consistency issue from a malicious receiver, and the challenge is how to resolve the issue while preserving the expected efficiency property. To this end, we propose new efficiency-preserving consistency checks, enabled by a set of new techniques, optimizations, and analyses. Combining the consistency checks with our framework based on authenticated correlations, we propose two maliciously secure secret-shared shuffle protocols with linear cost and constant-round communication. We have implemented our protocols. Performance evaluation shows that our protocols are faster with lower communication than the state-of-the-art.

Keywords: Secret-sharing · Shuffle · Correlation · Malicious security

X. Song and X. Liang—Co-first authors.

S. Bai and E. Persichetti (Eds.): PKC 2026, LNCS 16552, pp. 321–354, 2026.
https://doi.org/10.1007/978-3-032-26734-4_11

1 Introduction

Secret-shared shuffle (SSS) [6,13,23,39,53] permutes secret shared secrets using a random secret permutation, without revealing the shared secrets or the permutation. SSS is a basic primitive for secret-sharing-based secure computation, where shuffle is necessary to hide access patterns and linkability.

Shuffle is useful in secure data analytics applications. For example, shuffle is used to improve the efficiency of private database join via a "shuffle-then-filter" paradigm [2,25,33,40,45,56]: the computing parties obliviously shuffle secret-shared data items before revealing all shared records that match some criterion for the subsequent secure computation. Similarly, many protocols [3–5,11,29,30,40] leverage SSS to improve the efficiency of oblivious sorting: the parties shuffle data items first, followed by a more efficient non-oblivious secure comparison process. SSS is also a core primitive in anonymous communication systems with distributed trust. Mixnet-based anonymous communication [14] shuffles encrypted messages to break the linkability between messages and their owners. These systems employ public-key encryption and zero-knowledge proof of correct shuffling [8,49], leading to substantial computational overhead. Eskandarian and Boneh [23] proposed a system Clarion using SSS in the distributed-trust model, where non-colluded servers collaboratively shuffle messages shared between the servers and conduct integrity checks, achieving better efficiency than the mixnet-based approach. Other SSS-based anonymous communication systems [1,37,43] follow a similar paradigm. More applications using SSS includes private graph analysis [3,48,58], leakage suppression for encrypted search [41], and shuffle model of differential privacy [7,15,16,22,26,55].

Efficient SSS protocols benefit related protocols and applications. This paper focuses on efficient SSS protocols in the offline-online paradigm. Typically, protocols in the offline-online paradigm consist of an offline phase and an online phase. In the offline phase, the parties generate input-independent correlations that can be used to perform efficient secure computation in the online phase when the inputs are ready. The CGP shuffle protocol [13] proposed by Chase, Ghosh, and Poburinnay is an efficient candidate. CGP shuffle consists of two phases. The parties first generate pseudorandom correlations in the offline phase and then use the correlations to conduct a highly efficient shuffle phase, requiring only constant-round and linear online communication. Considering its promising efficiency properties, many recent works [2,6,23,31,33,38,39,41,43,50,56] leverage or adapt CGP shuffle to various data-oblivious protocols and applications. For example, TikTok implements CGP shuffle in its PETAce library.[1]

The original CGP shuffle is only semi-honest secure, which assumes the parties follow the protocol faithfully. Semi-honest security could be trivially broken under malicious deviations, which may not be sufficient for highly sensitive applications [23,55,56], where malicious security is desirable to provide realistic security guarantees. How to enhance CGP shuffle with malicious security while maintaining acceptable efficiency is non-trivial. Existing protocols [23,39,53]

[1] https://github.com/tiktok-privacy-innovation/PETAce.

attempted to achieve this challenging goal. Among them, the protocols from [23] and [39] are subject to privacy attacks as analysed by [53]. Song *et al.* [53] further proposed an SSS protocol, which is the only existing maliciously secure CGP-like shuffle protocol over authenticated linear secret sharing (ALSS).

The protocol from [53] relies on two core components to achieve malicious security. First, it proposes lightweight correlation checks to ensure all generated correlations are well-formed, defeating attacks to previous works [23,39] that exploit non-well-formed correlations. Second, it directly applies the semi-honest CGP shuffle over ALSS to conduct a maliciously secure shuffle, followed by a folklore integrity check method [23,39] based on message authentication code (MAC), which is commonly used in SPDZ-family protocols [20,36] to check the correctness of secure computation over ALSS.

While these well-formedness checks ensure integrity and correctness in the presence of malicious deviations, *the shuffle phase of all previous works* [23,39,53] *is not fully authenticated*, which can be exploited to break privacy. We provide a sketched description and refer to the main body for details. In particular, the previous "CGP shuffle over ALSS" solutions cannot check the well-formedness of the protocol messages immediately, and they can only resort to a post-execution MAC check over the resulting ALSS secrets that are computed from the possibly non-well-formed messages. Song *et al.* [53] observed that such a paradigm allows a malicious sender to breach *privacy* via selective failure attacks, where a malicious sender can inject errors into its protocol messages and the protocol still completes normally if he/she correctly guessed some information about the secret permutation; in this case, the sender learns the secret information about the receiver's permutation without being detected. To obtain full privacy, Song *et al.* [53] proposed a cut-and-choose leakage reduction mechanism via repeated shuffling to remove possible leakage. However, repeated shuffling destroys the linear-cost and constant-round-communication property of the original semi-honest CGP shuffle.

Previous roadmaps failed to achieve linear cost and constant-round communication for maliciously secure secret-shared shuffle over ALSS. To our best knowledge, this holds beyond the CGP shuffle: no existing SPDZ-family maliciously secure secret-shared shuffle protocol achieves this property.

Solving this problem is not as trivial as one may expect, and we are not the only attempt. A more recent work [24] proposes an SSS protocol with claimed malicious security and linear online communication, aiming to avoid the efficiency limitation from [53]. Our security analysis shows that [24] still suffers from the same type of selective failure attacks, and we demonstrate with a concrete example. It is unclear how to upgrade [24] to be truly maliciously secure while maintaining its claimed efficiency. We provide more details in Sect. 6.

Existing attempts and failures demonstrate that the technical roadmap applying unauthenticated CGP shuffle over ALSS has inherent limitations in security and efficiency. Hence, we tend to believe that solving this problem with our expected efficiency properties would require a new roadmap.

Roadmap and Contribution. We propose MOSAC, a new suite of maliciously secure two-party SPDZ-family SSS protocols with linear cost and constant-round communication in the offline-online preprocessing paradigm. We implement MOSAC and report concrete performance. MOSAC is around 6× faster in the shuffle phase than [53] when shuffling an ALSS vector of dimension $n = 2^{20}$. We achieve the goal with the following new roadmap and techniques.

New Shuffle Paradigm. We propose a new framework for designing maliciously secure shuffle protocols based on authenticated correlations. We first propose an authenticated correlation called *authenticated shuffle tuple (AST)*. Based on that, we design MOSAC in the *arithmetic black box* (ABB) fashion, making the protocol easier for modular security abstraction.

Our protocol differs from previous solutions [23,39,53]. Our solution permits no shuffle-phase selective failure attacks from a malicious sender. This property avoids repeated online execution in the previous roadmap [53]. In particular, it is much easier for MOSAC to maintain full privacy, and most of the correctness issues can be efficiently addressed by existing authentication mechanisms from ALSS in a conceptually simple black-box fashion.

Efficiency-Preserving Consistency Checks. While all sounds good, our roadmap introduces a new consistency issue from a malicious receiver. In particular, our protocol will temporarily "*leave*" the authenticated world in a critical protocol step. Looking ahead, this step requires the parties to open a masked vector to the receiver who holds the permutation, and requires the receiver to faithfully permute the masked vector and reshare the permuted masked vector back to the authenticated world. Since the operation is locally conducted by the receiver, the receiver may not abide by the protocol, and he/she may provide arbitrary values to break correctness (and privacy, as we will show).

Maliciously secure protocols typically rely on consistency checks to enforce honest behaviors, which check whether the parties provide consistent and expected values according to the protocol specification. Depending on scenarios, consistency checks can be instantiated via zero-knowledge proof, cut-and-choose mechanisms [42], homomorphic commitments/MACs [19–21,35], etc. The challenge lies in minimizing the introduced overhead. We aim to preserve the desired efficiency properties even after applying these checks.

We propose new *efficiency-preserving* consistency checks without breaking the expected efficiency. Our first consistency check applies a classic polynomial-based permutation check [39,49] to check whether two authenticated vectors satisfy a permutation relation. We formally prove that conducting two permutation checks is sufficient to enforce honest receiver-side behaviors.

However, trivially evaluating the permutation check requires at least $O(\log n)$ rounds of communication, which destroys our expected efficiency goal. To resolve the issue, we propose a new evaluation protocol with linear cost and constant-round online communication. At the core of the protocol is an efficient n-input multiplication protocol with constant rounds and linear cost. Security of this optimization requires some care, and we provide a formal security proof and analysis. The proposed protocols and analysis may be of independent interest.

We further propose a simpler *double-authentication* consistency check protocol that leverages another layer of authentication to enforce honest behaviors, which we call *layer-two* authentication. As an interesting property, it ensures that the corrupted receiver must follow the protocol specification even when the (layer-one) SPDZ MAC is taken off, and inconsistency can still be detected via the remaining layer-two authentication. Compared to the consistency check from the permutation check, our layer-two authentication is much simpler and avoids evaluating the relatively expensive permutation-check polynomials. Overall, it still requires linear cost and constant-round communication. The idea of double authentication may be helpful in the design of other oblivious protocols.

Combining AST, authentication mechanisms for ALSS, consistency checks, and customized optimizations, we propose two SSS protocols parameterized from two consistency check strategies, both achieving linear cost and constant-round communication.

Authenticated Correlation Preprocessing. ASTs can be preprocessed independently of the input in the preprocessing phase, which is a common paradigm for SPDZ-family protocols. While there could be several approaches for AST preprocessing, we provide an approach by adapting [53]. Our tailored preprocessing builds ASTs from unauthenticated CGP correlations and random authenticated secret sharing. Similar to previous works [23,39,53], the preprocessing protocol is leaky in the sense that a malicious sender could perform selective failure attacks, allowing him/her to learn sensitive information about the secret permutation with a small but non-negligible probability. So we employ the cut-and-choose leakage reduction mechanism from [53] to generate non-leaky and well-formed AST correlations. We additionally propose new optimizations to the correlation preprocessing to improve concrete efficiency.

Implementation and Performance. We have implemented necessary primitives and protocols for maliciously secure secret-shared shuffle, including SPDZ preprocessing, correlation generation, and associated malicious security mechanisms. We hope our open-source implementation[2] will facilitate future research in related fields. We implement MOSAC and [53] based on the codebase. The concrete performance evaluation shows that MOSAC is around 6× faster than [53] when shuffling $n = 2^{20}$ ALSS secrets.

Summary of Contribution

- A new and conceptually simple framework for maliciously secure two-party secret-shared shuffle protocols based on authenticated correlations.
- New efficiency-preserving consistency checks to enforce honest behaviors. Efficiency-preserving is achieved by a set of new observations, protocols, and analyses, which may be of independent interest.
- Modularly combining the proposed techniques results in the first two-party maliciously secure SPDZ-family secret-shared shuffle with linear cost and constant-round communication in the preprocessing model.

[2] https://github.com/liang-xiaojian/MOSAC.

- We implement our protocol and open-source the code. Concrete performance evaluation shows our protocol is 6× faster than the state-of-the-art.

Organization. We present preliminary in Sect. 2. Section 3 overviews our roadmap, and we propose two maliciously secure MOSAC shuffle protocols in Sect. 4 and Sect. 5, respectively. Section 6 presents our attack to [24]. We present implementation and performance evaluation in Sect. 7. Section 8 shows related works and we conclude in Sect. 9.

2 Preliminary

2.1 Notations

We use λ and κ to denote the statistical and computational security parameters, respectively, and $\mathsf{negl}(\cdot)$ denotes a negligible function. We use $[n]$ to denote the set $\{0, 1, \ldots, n-1\}$ and $[a, b]$ to denote $\{a, a+1, \ldots, b-1, b\}$. We use the bold lower-case letter $\boldsymbol{a}$ to denote a vector and a_i as its i-th element, and the bold uppercase letter $\mathbf{A}$ to represent a matrix and $\mathbf{A}_{i,j}$ denotes its element at row i and column j. We use $x \leftarrow y$ to denote value assignment from y to x, and $x \xleftarrow{\$} \mathcal{D}$ denotes that x is uniformly sampled from a set $\mathcal{D}$. We use $\mathbb{F}$ to denote a finite field, and $\mathbb{D}$ to denote some domain such as $\mathbb{D} = \mathbb{F}^2$. We use $\mathsf{Uniform}(\mathbb{D})$ to denote the uniform distribution over $\mathbb{D}$. Let D_1 and D_2 be two distributions, we use $\mathsf{SD}(D_1, D_2)$ to denote the statistical distance between D_1 and D_2. Given two random variables $X \sim D_1$ and $Y \sim D_2$ following distributions D_1 and D_2, respectively, we may abuse the notation $\mathsf{SD}(X, Y)$ to denote the statistical distance between the distributions that X and Y follow.

Permutation. An n-swap permutation π is a bijective function $\pi : [n] \mapsto [n]$. We use $\mathbf{S}_n$ to denote the symmetric group containing all n-swap permutations. Applying π over a vector $\boldsymbol{x}$, we define

$$\boldsymbol{y} = \pi(\boldsymbol{x}) = (x_{\pi(0)}, \cdots, x_{\pi(n-1)}), \tag{1}$$

where $y_i = x_{\pi(i)}$ for $i \in [n]$. Namely, $x_{\pi(i)}$ is moved to position i after applying the permutation. We denote π^{-1} as the inverse of a permutation π, and $\pi_1 \circ \pi_0$ as the composition of permutations π_1 and π_0. When applying $\pi_1 \circ \pi_0$ over $\boldsymbol{x}$, we define $\pi_1 \circ \pi_0(\boldsymbol{x}) = \pi_1(\pi_0(\boldsymbol{x}))$, which we call a *composed* permutation. In general, we define $\pi_{B-1} \circ \cdots \circ \pi_1 \circ \pi_0(\boldsymbol{x}) = \pi_{B-1}(\cdots \pi_1(\pi_0(\boldsymbol{x})))$ for $B \geq 2$.

2.2 Secret Sharing

Linear Secret Sharing. Let $[\![x]\!]$ denote an additive linear secret sharing (LSS) of $x \in \mathbb{F}$ shared between k parties, where P_i holds a share $[\![x]\!]_i \in \mathbb{F}$ such that $\sum_{i \in [k]} [\![x]\!]_i = x$. The parties can reveal x by publishing all shares. LSS supports affine transformation using public $a, b, c \in \mathbb{F}$ via local computation over shares:

- $[\![z]\!] \leftarrow a \cdot [\![x]\!] + b \cdot [\![y]\!] + c$: P_0 computes $[\![z]\!]_0 \leftarrow a \cdot [\![x]\!]_0 + b \cdot [\![y]\!]_0 + c$. P_i computes $[\![z]\!]_i \leftarrow a \cdot [\![x]\!]_i + b \cdot [\![y]\!]_i$ for $i \in [k] \setminus \{0\}$.

Authenticated Linear Secret Sharing. Authenticated linear secret sharing (ALSS) additionally ensures the integrity of shared secrets, which is usually used for secure computation in the presence of malicious adversaries. A SPDZ-style ALSS binds an LSS secret $[\![x]\!]$ with an information-theoretic MAC sharing $[\![\gamma(x)]\!]$, where $\gamma(x) = \alpha \cdot x$ and $\alpha \in \mathbb{F}$ is a secret global MAC key shared between the parties. Let $\langle x \rangle = ([\![x]\!], [\![\gamma(x)]\!])$ denote an ALSS of x where P_i holds an *authenticated share* $\langle x \rangle_i = ([\![x]\!]_i, [\![\gamma(x)]\!]_i) \in \mathbb{D}$; here $\mathbb{D} = \mathbb{F}^2$. Similar to LSS, ALSS supports the following affine computation:

- $\langle z \rangle \leftarrow a \cdot \langle x \rangle + b \cdot \langle y \rangle + c$: the parties compute $\langle z \rangle \leftarrow (a \cdot [\![x]\!] + b \cdot [\![y]\!] + c, a \cdot [\![\gamma(x)]\!] + b \cdot [\![\gamma(y)]\!] + c \cdot [\![\alpha]\!])$.

ALSS uses MACs to detect errors, with soundness error proportional to $1/|\mathbb{F}|$. We require $\mathbb{F}$ is sufficiently large (*i.e.*, $|\mathbb{F}| > 2^\lambda$) to obtain overwhelming detection probability. In this paper, we use $[\![\boldsymbol{x}]\!]$, $[\![\gamma(\boldsymbol{x})]\!]$, and $\langle \boldsymbol{x} \rangle = ([\![\boldsymbol{x}]\!], [\![\gamma(\boldsymbol{x})]\!])$ to denote an LSS vector sharing of $\boldsymbol{x}$, LSS vector sharing of $\gamma(\boldsymbol{x})$, and ALSS vector sharing of $\boldsymbol{x}$, respectively, where $\gamma(x_i) = \alpha \cdot x_i$.

Functionality $\mathcal{F}^{\langle \cdot \rangle}$ [36, 20, 19]

Parameters: Dictionary Val storing all secrets.

- **Rand**: on input $(\mathsf{rand}, \mathsf{id})$ from all parties, sample $r \xleftarrow{\$} \mathbb{F}$ and store $\mathsf{Val}[\mathsf{id}] \leftarrow r$.
- **Input**: on input $(\mathsf{input}, i, \mathsf{id}, x)$ from P_i and $(\mathsf{input}, i, \mathsf{id})$ from all other parties, store $\mathsf{Val}[\mathsf{id}] \leftarrow x$.
- **Triple**: on input $(\mathsf{triple}, \mathsf{id}_a, \mathsf{id}_b, \mathsf{id}_c)$ from all parties, sample $a, b \xleftarrow{\$} \mathbb{F}$. Store $(\mathsf{Val}[\mathsf{id}_a], \mathsf{Val}[\mathsf{id}_b], \mathsf{Val}[\mathsf{id}_c]) \leftarrow (a, b, a \cdot b)$.
- **LinCom**: on input $(\mathsf{lincom}, \mathsf{id}_x, \mathsf{id}_y, \mathsf{id}_z, a, b, c)$ from all parties, compute $z = a \cdot \mathsf{Val}[\mathsf{id}_x] + b \cdot \mathsf{Val}[\mathsf{id}_y] + c$. Store $\mathsf{Val}[\mathsf{id}_z] \leftarrow z$.
- **Mul**: on input $(\mathsf{mul}, \mathsf{id}_x, \mathsf{id}_y, \mathsf{id}_z)$ from all parties, compute $z = \mathsf{Val}[\mathsf{id}_x] \cdot \mathsf{Val}[\mathsf{id}_y]$. Store $\mathsf{Val}[\mathsf{id}_z] \leftarrow z$.
- **OpenCheck**: on input $(\mathsf{opencheck}, \mathsf{id})$ from all parties, send $\mathsf{Val}[\mathsf{id}]$ to $\mathcal{A}$ and wait x from $\mathcal{A}$. Output x to all parties. Wait for further instructions from $\mathcal{A}$. If $\mathcal{A}$ inputs OK and $x = \mathsf{Val}[\mathsf{id}]$, return OK to all parties. Otherwise, abort.

Fig. 1. Ideal functionality for ALSS.

ALSS Functionalities. Figure 1 presents an ALSS functionality $\mathcal{F}^{\langle \cdot \rangle}$. $\mathcal{F}^{\langle \cdot \rangle}$ supports several available ideal commands (*i.e.*, sub-functionalities) for ALSS in the *Arithmetic Black-Box* (ABB) model, which stores secrets in a dictionary Val and conducts operations over them. This functionality allows evaluating basic operations on values such as addition and multiplication in a black-box storage while

preserving their context. The ABB model is commonly used in secret-sharing-based MPC, enabling a clean and modular protocol description. These commands can be securely realized from existing well-established SPDZ-family protocols [19,21,35]. For convenience, we use functionality $\mathcal{F}_{\mathsf{cmd}}^{\langle\cdot\rangle}$ for a concrete command cmd supported by $\mathcal{F}^{\langle\cdot\rangle}$. For example, $\mathcal{F}_{\mathsf{Mul}}^{\langle\cdot\rangle}$ denotes the sub multiplication functionality. In existing formal functionality definitions for ALSS [19–21,36], an ALSS secret x should be opened using a command Open when necessary (*e.g.*, opening the final computation result). To defeat *additive errors* [27] from the adversary during the opening, the command Check [36] is used to detect incorrect opening. Since Open is usually followed by command Check to enforce correct opening in the malicious setting, we instead use a new command $\mathsf{OpenCheck}$ to combine them for simplicity.

ALSS sharing $(\langle a\rangle, \langle b\rangle, \langle a \cdot b\rangle)$ generated by command Triple is known as a Beaver's multiplication triple (MT) [9]. An MT can be used for efficiently implementing $\mathcal{F}_{\mathsf{Mul}}^{\langle\cdot\rangle}$. Suppose the parties pre-share an MT $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, they can compute $\langle x \cdot y\rangle$ from $\langle x\rangle$ and $\langle y\rangle$ as follows:

- The parties compute $\langle e\rangle \leftarrow \langle x\rangle - \langle a\rangle$ and $\langle f\rangle = \langle y\rangle - \langle b\rangle$.
- The parties open $\langle e\rangle$ and $\langle f\rangle$ via $\mathcal{F}_{\mathsf{OpenCheck}}^{\langle\cdot\rangle}$. Abort if the open or check fails.
- The parties compute $\langle z\rangle \leftarrow f \cdot \langle a\rangle + e \cdot \langle b\rangle + \langle c\rangle + e \cdot f$.

We use $\langle z\rangle \leftarrow \langle x\rangle \cdot \langle y\rangle$ to denote secret-shared multiplication.

3 Problem Statement

3.1 Design Goals

Ideal Functionality. MOSAC focuses on the two-party secret-shared shuffle. MOSAC performs SSS over an input ALSS vector $\langle \boldsymbol{x}\rangle$ shared between $\mathtt{S}$ and $\mathtt{R}$ and outputs an ALSS vector $\langle \boldsymbol{y}\rangle$ such that $\boldsymbol{y} = \pi(\boldsymbol{x})$ for a random secret permutation π. In the presence of malicious adversaries with static corruption, MOSAC maintains *privacy* in any case and *correctness* if the protocol completes without abort (*i.e.*, *malicious security with abort*). Informally, privacy requires that the resulting SSS protocol ensures the privacy of $\boldsymbol{x}$, $\boldsymbol{y}$, and π. Correctness requires the integrity of ALSS secrets $\langle \boldsymbol{x}\rangle, \langle \boldsymbol{y}\rangle$ and correct shuffling. The security holds if the adversary statically corrupts any one party. Security goals are formally captured by an ideal SSS functionality $\mathcal{F}_{\mathsf{sss}}$ in Fig. 2.

In this paper, we instead securely realize an *One-sided Shuffle* (OSS) functionality $\mathcal{F}_{\mathsf{oss}}^{[\mathtt{R}]}$ [53]: the sender $\mathtt{S}$ and receiver $\mathtt{R}$ jointly share an ALSS vector $\langle \boldsymbol{x}\rangle$ and $\mathtt{R}$ additionally provides a permutation $\pi \in \mathbf{S}_n$. $\mathcal{F}_{\mathsf{oss}}^{[\mathtt{R}]}$ computes $\boldsymbol{y} = \pi(\boldsymbol{x})$ and stores $\boldsymbol{y}$. Here we omit the formal description of $\mathcal{F}_{\mathsf{oss}}^{[\mathtt{R}]}$, as the only difference between $\mathcal{F}_{\mathsf{oss}}^{[\mathtt{R}]}$ and $\mathcal{F}_{\mathsf{sss}}^{[\mathtt{R}]}$ is that the receiver $\mathtt{R}$ in $\mathcal{F}_{\mathsf{oss}}^{[\mathtt{R}]}$ learns the permutation π, while π in $\mathcal{F}_{\mathsf{sss}}^{[\mathtt{R}]}$ is hidden from both parties. We note that OSS is sufficient for designing SSS via a reversed execution strategy [13,39,53] – by simply sequentially invoking OSS twice with the roles of sender and receiver reversed, where

Functionality $\mathcal{F}_{\mathsf{sss}}$

Parameters: $n \in \mathbb{N}$; a dictionary Val storing all ALSS secrets.
Functionality: upon receiving $(\mathsf{sss}, \{\mathsf{id}_i\}_{i\in[n]}, \{\mathsf{id}'_i\}_{i\in[n]})$ from $\mathtt{R}$ and $\mathtt{S}$:

- Sample $\pi \xleftarrow{\$} \mathbf{S}_n$.
- Fetch the vector $\boldsymbol{x}$ such that $\boldsymbol{x}_i = \mathsf{Val}[\mathsf{id}_i]$ for $i \in [n]$.
- Compute $\boldsymbol{y} = \pi(\boldsymbol{x})$. Store $\mathsf{Val}[\mathsf{id}'_i] \leftarrow \boldsymbol{y}_i$ for $i \in [n]$.

Fig. 2. Functionality for secret-shared shuffle.

the second OSS is applied over the shuffled ALSS vector from the first OSS output. Therefore, we focus on how to realize $\mathcal{F}_{\mathsf{oss}}^{[\mathtt{R}]}$ in this paper.

Efficiency Goal. As mentioned, our goal is to design maliciously secure two-party SSS protocols with $O(n)$ communication in $O(1)$ rounds, and $O(n)$ computation complexity in the shuffle phase.

3.2 Previous Roadmaps

Semi-Honest CGP Shuffle. The original semi-honest CGP shuffle achieves linear cost and constant-round communication. In the preprocessing phase, the parties generate shuffle correlation such that the sender $\mathtt{S}$ obtains two vectors $(\boldsymbol{a}, \boldsymbol{b})$ and the receiver $\mathtt{R}$ obtains a permutation π and a vector $\boldsymbol{\Delta} = \pi(\boldsymbol{a}) - \boldsymbol{b}$. In the shuffle phase, the parties run as follows.

1. $\mathtt{S}$ simply sends $\boldsymbol{\delta} = \boldsymbol{x} - \boldsymbol{a}$ to $\mathtt{R}$.
2. $\mathtt{S}$ sets $[\![\boldsymbol{y}]\!]_{\mathtt{s}} = \boldsymbol{b}$ and $\mathtt{R}$ sets $[\![\boldsymbol{y}]\!]_{\mathtt{r}} = \pi(\boldsymbol{\delta}) + \boldsymbol{\Delta} = \pi(\boldsymbol{x}) - \boldsymbol{b}$.

One can check that $[\![\boldsymbol{y}]\!]_{\mathtt{s}} + [\![\boldsymbol{y}]\!]_{\mathtt{r}} = \pi(\boldsymbol{x})$ as requried. We use $(\!|\pi|\!) = ((\pi, \boldsymbol{\Delta}), (\boldsymbol{a}, \boldsymbol{b}))$ to denote a *shuffle tuple* correlation used in the CGP shuffle, where $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{\Delta} \in \mathbb{D}^n$ for some domain $\mathbb{D}$. Here we have $\mathbb{D} = \mathbb{F}$ for LSS defined over $\mathbb{F}$; we can define shuffle tuple over other domains if necessary, *e.g.*, $\mathbb{D} = \mathbb{F}^2$. The CGP shuffle enjoys a highly efficient shuffle phase, incurring linear communication in a constant round, and the parties conduct cheap local computation.

Maliciously Secure CGP Shuffle. Aiming to enhance CGP shuffle with malicious security while minimizing the introduced overhead, three existing works [23,39,53] attempted to design maliciously secure CGP shuffle protocols. Among them, the protocols from Eskandarian and Boneh [23] and Laud [39] are subject to privacy flaws as analyzed by Song *et al.* [53]. Song *et al.* further proposed a maliciously secure CGP shuffle, but it achieves malicious security by increasing online overhead. Now we revisit this line of work with more details.

Selective Failure Attacks. Existing works [23,39,53] apply OSS protocol over ALSS vector $\langle \boldsymbol{x} \rangle$ to perform shuffle, with the help of a pre-computed shuffle tuple $(\!|\pi|\!) = ((\pi, \boldsymbol{\Delta}), (\boldsymbol{a}, \boldsymbol{b})) \in (\mathbf{S}_n \times \mathbb{D}^n) \times (\mathbb{D}^n \times \mathbb{D}^n)$, where $\mathbb{D} = \mathbb{F}^2$ to be compatible with authenticated shares (recall that each ALSS share can be interpreted as an element over $\mathbb{F}^2$). We sketch this protocol as follows:

1. S sends $\boldsymbol{\delta} = \langle \boldsymbol{x} \rangle_{\mathsf{s}} - \boldsymbol{a}$ to R. S sets $\langle \boldsymbol{y} \rangle_{\mathsf{s}} \leftarrow \boldsymbol{b}$.
2. R receives $\boldsymbol{\delta}$ and sets $\langle \boldsymbol{y} \rangle_{\mathsf{r}} \leftarrow \pi(\langle \boldsymbol{x} \rangle_{\mathsf{r}} + \boldsymbol{\delta}) + \boldsymbol{\Delta}$.
3. Run a post-execution batch MAC check protocol Π_{MacCheck} over $\langle \boldsymbol{y} \rangle$ to detect errors:
   a) Call $\{c_i\}_{i \in [n]} \leftarrow \mathcal{F}_{\mathsf{Coin}}(\mathbb{F}^n)$ and $\langle r \rangle \leftarrow \mathcal{F}_{\mathsf{Rand}}^{\langle \cdot \rangle}(\mathbb{F})$
   b) $\langle t \rangle \leftarrow \sum_{i \in [n]} c_i \cdot \langle x_i \rangle + \langle r \rangle$.
   c) $t \leftarrow \mathcal{F}_{\mathsf{OpenCheck}}^{\langle \cdot \rangle}(\langle t \rangle)$. If $\mathcal{F}_{\mathsf{OpenCheck}}^{\langle \cdot \rangle}$ aborts, abort.
4. Return $\langle \boldsymbol{y} \rangle$ if the protcol does not abrot.

The above protocol essentially applies CGP shuffle over an ALSS vector. Assuming the parties behave faithfully, we have $\langle \boldsymbol{y} \rangle_{\mathsf{r}} + \langle \boldsymbol{y} \rangle_{\mathsf{s}} = (\boldsymbol{y}, \gamma(\boldsymbol{y}))$, which means the parties share an authenticated vector $\langle \boldsymbol{y} \rangle$. To enforce integrity, the parties run a standard post-execution MAC check without revealing $\boldsymbol{y}$. Roughly, this is done by opening a random linear combination of $\{\langle y \rangle_i\}_{i \in [n]}$, added with a mask $\langle r \rangle$, and conduct a standard MAC check, where $\{c_i\}_{i \in [n]}$ are random coins sampled via coin-tossing functionality $\mathcal{F}_{\mathsf{Coin}}$, $\langle r \rangle$ is generated by $\mathcal{F}_{\mathsf{Rand}}^{\langle \cdot \rangle}$ serving as a mask, thus the check reveals nothing about $\boldsymbol{y}$ except its integrity status.

Though the post-execution check enforces correctness, Song *et al.* [53] show that a malicious sender can exploit the check to breach privacy via selective failure attacks, revealing secret information about π with non-negligible probability. We sketch the attack in Fig. 3. In this attack, the malicious sender S samples a one-hot error vector $\boldsymbol{e} \in \mathbb{D}^n$ with the non-zero element appearing at position q. Following the protocol specification, this non-zero error will be permuted by π to position $\pi^{-1}(q)$. If S still sets $\langle \boldsymbol{y} \rangle_{\mathsf{s}} = \boldsymbol{b}$, certainly the introduced error will break the MAC relation, which will be detected by the following MAC check.

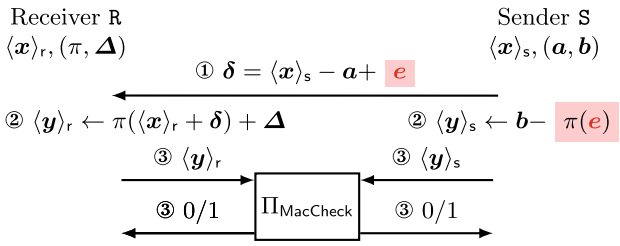

Fig. 3. Selective failure attacks [53]. The malicious S injects error $\boldsymbol{e}$ to the protocol message in step ① and removes $\pi(\boldsymbol{e})$ from its own share before MAC check.

However, since $\boldsymbol{e}$ is a one-hot vector, there are only n possible $\pi(\boldsymbol{e})$ after shuffling, thus S can guess $\pi(\boldsymbol{e})$ and remove the *permuted* error from its local share $\langle \boldsymbol{y} \rangle_{\mathsf{s}}$ *before* running the check. Note that the MAC check requires the parties to commit to their checking messages first, thus there will be two outcomes depending on whether the adversary guessed correctly. If S guessed correctly with probability $1/n$, the parties would still share a well-formed ALSS vector so that the check would complete normally. Otherwise, S would be caught due to

the introduced errors, with probability $1 - 1/n$. The above attack is rooted in the *unauthenticated* nature of the protocol message $\boldsymbol{\delta}$, so S can freely inject and remove errors, and the post-execution check serves as an *oracle* to tell whether this attack is successful or not; this is essentially a selective-failure attack. Note that S may add errors to more positions in the selective failure attack, but he will be caught with higher probability. Song *et al.* [53] also demonstrates that the offline phase of previous works [23] also suffers from selective failure attacks.

Leakage Reduction and its Overhead. Both the offline and online phases of [53] are subject to selective failure attacks. Song *et al.* further propose a *shuffle-phase* cut-and-choose-based leakage reduction mechanism to remove two-phase leakages. The intuitive idea is to conduct repeated shuffles, such that an input ALSS vector is permuted by sufficient permutations, and the parties run integrity checks for each sub-shuffle. If the malicious S dares to learn any information about the composed permutation $\pi_{B-1} \circ \pi_{B-2} \circ \cdots \circ \pi_0$, he/she must attack each involved permutation. Since the adversary can only win each attack with a small probability, the adversary who dares to attack all permutations will be detected with an overwhelming probability for a sufficiently large B. Hence, if B is large enough and the protocol does not abort, R has overwhelming confidence that there exists at least one non-leaky permutation such that $\pi_{B-1} \circ \pi_{B-2} \circ \cdots \circ \pi_0$ is still a uniformly random permutation in the view of S. However, repeated execution increases shuffle-phase overhead, which destroys the original efficiency property from the semi-honest CGP shuffle. The overhead seems inherent following the roadmap from [53] – *the unauthenticated and leaky online phase necessitates repeated execution to remove possible leakage.*

3.3 Our Roadmap

Authenticated Shuffle Tuple. The online phase of previous works [23,39,53] simply applies the (semi-honest) CGP shuffle over an ALSS vector in a non-ABB fashion, followed by a lightweight batched MAC check. We acknowledge that such a non-ABB design is mainly for reducing the overhead by bootstrapping the efficiency property from the CGP shuffle. However, the price is that an honest receiver cannot directly check the well-formedness of the shuffle-phase protocol messages from the sender. A malicious sender can manipulate the protocol messages by injecting selective-failure errors, leaving attacks that reveal sensitive information about the receiver's permutation.

MOSAC takes a different strategy by applying authenticated correlation directly over ALSS, so that we can design consistency checks by bootstrapping SPDZ mechanisms in an ABB fashion. Looking ahead, our new design eliminates shuffle-phase selective failure attacks from a malicious sender easily.

Specifically, MOSAC relies on a new authenticated correlation called *authenticated shuffle tuple* (AST). In an AST correlation

$$\langle\!\langle \pi \rangle\!\rangle = (\pi, \langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle),$$

the receiver specifies a permutation $\pi \in \mathbf{S}_n$ and two ALSS vectors $\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle$ are shared between the parties, where $\boldsymbol{b} = \pi(\boldsymbol{a})$ and $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{D}^n$; we can define $\mathbb{D} = \mathbb{F}^k$

for some $k \in \mathbb{N}$ to allow a k-column shuffle. Informally, we say an AST is *well-formed* if it is correctly correlated as desired, and it is *non-leaky* if (1) the parties learn no more information about $\boldsymbol{a}$ and $\boldsymbol{b}$, and (2) the sender learns no more information about π. AST is authenticated as we incorporate SPDZ MACs within AST, which will be used to enforce honest execution when the correlation is used in our AST-based shuffle protocols.

Shuffle from AST. AST supports a conceptually simple, efficient, and, more importantly, authenticated shuffle protocol. Suppose the parties want to shuffle $\langle \boldsymbol{x} \rangle$ with the help of a non-leaky and well-formed AST $\langle\!\langle \pi \rangle\!\rangle$, then we have an AST-based OSS protocol sketched in Fig. 4.

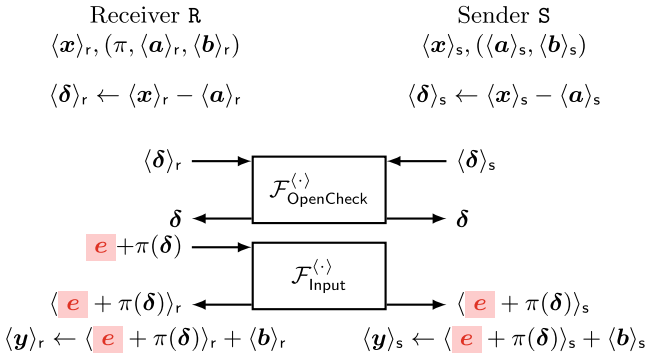

Fig. 4. The sketched AST-based OSS protocol.

The parties compute $\langle \boldsymbol{\delta} \rangle = \langle \boldsymbol{x} \rangle - \langle \boldsymbol{a} \rangle$ and open $\langle \boldsymbol{\delta} \rangle$ using $\mathcal{F}^{\langle \cdot \rangle}_{\mathsf{OpenCheck}}$. Next, R locally computes $\pi(\boldsymbol{\delta})$ and reshares $\langle \pi(\boldsymbol{\delta}) \rangle$ between the parties, and we call this step the *local-permute-and-share* step. The parties then computes $\langle \boldsymbol{y} \rangle = \langle \pi(\boldsymbol{\delta}) \rangle + \langle \boldsymbol{b} \rangle$. If all parties behave honestly, then we can check

$$\langle \boldsymbol{y} \rangle = \langle \pi(\boldsymbol{\delta}) \rangle + \langle \boldsymbol{b} \rangle = \langle \pi(\boldsymbol{x}) \rangle - \langle \pi(\boldsymbol{a}) \rangle + \langle \boldsymbol{b} \rangle = \langle \pi(\boldsymbol{x}) \rangle.$$

The last equality holds since $\boldsymbol{b} = \pi(\boldsymbol{a})$ following the well-formedness of AST. Note that opening $\boldsymbol{\delta}$ reveals no more information about $\boldsymbol{x}$, as $\boldsymbol{a}$ is uniformly sampled from $\mathbb{F}^n$ and hidden from both parties. It is not hard to see that the protocol is secure against semi-honest adversaries.

Unlike semi-honest adversaries who faithfully follow the protocol specification, a malicious adversary may take arbitrary strategies to undermine correctness and/or privacy. In particular, it may inject errors during the protocol to create inconsistencies, which could be further exploited to break privacy (*e.g.*, via selective-failure attacks). We must enforce honest behaviors on both sides, so that the parties can be sure that the protocol proceeds as expected.

TODO: Enforce Sender-Side Honest Behaviors. It is worth noticing that the sender-side operations are fully designed in an ABB fashion. The sender's

malicious activities are limited, and it is relatively straightforward to enforce sender-side honest behavior by bootstrapping SPDZ authentication mechanisms in a black-box fashion. With that, a corrupted sender cannot conduct shuffle-phase selective failure attacks as in previous works [23,39,53]. The key difference lies in the fact that prior strategies lacked authentication of the sender's shuffle-phase messages, thereby enabling selective-failure attacks. Our design methodology behind MOSAC follows a different strategy, allowing easier handling of malicious sender-side behaviors, which is the first key feature of our roadmap.

TODO: Enforce Receiver-Side Honest Behaviors. Enforcing receiver-side behavior is more challenging for our roadmap. While most receiver-side steps can be authenticated via existing authentication mechanisms from ALSS, the local-permute-and-share step requires the receiver to faithfully permute the opened vector $\boldsymbol{\delta}$ using the *exact* permutation π from the used AST, and reshare the expected permuted vector $\pi(\boldsymbol{\delta})$ back to the authenticated world. However, a malicious receiver may provide any vector following arbitrary strategies. Equivalently, this corresponds to adding an arbitrary error as depicted in Fig. 4. How to enforce receiver-side faithful behavior without destroying the expected efficiency properties remains a challenge. To this end, we propose two efficiency-preserving consistency checks to enforce receiver-side honest behaviors, serving as the second key feature.

Preprocessing AST. This paper focuses on designing authenticated shuffle protocols based on authenticated correlations. Though any secure OSS protocol can be used to preprocess AST over random ALSS sharing $\langle \boldsymbol{a} \rangle$, we provide an AST preprocessing protocol by adapting the cut-and-choose strategy from [53], and propose new optimizations to improve efficiency concretely, some of which can be applied back to [53]. The main body of this paper focuses on designing authenticated SSS from *non-leaky* and *well-formed* ASTs. Preprocessing ASTs is not the focus of this paper, so we move the AST preprocessing to the full version. In this paper, we will resort to an ideal AST generation functionality $\mathcal{F}_{\mathsf{GenAST}}$ for AST preprocessing.

We stress that the security of our online protocol is not inherently tied to [53], since other malicious-secure pre-processing protocols [34] can be used to realize the AST preprocessing functionality, and any future improvement can benefit the AST preprocessing. As a promising direction, recent work [50] proposes more concretely efficient CGP correlation preprocessing in the semi-honest setting. It remains to design malicious-secure correlation preprocessing for [50], and to see whether it achieves better efficiency than our malicious-secure preprocessing following [53], which we leave as an interesting direction for future work.

4 MOSAC via Permutation Check

4.1 High-Level Idea

It is not hard to see that all ABB operations in Fig. 4 can be fully authenticated via existing malicious-security SPDZ mechanisms. The remaining challenge lies

in ensuring that the receiver correctly inputs $\pi(\boldsymbol{\delta})$ back to the authenticated world. Before presenting our solution, we observe that the following two conditions fully defines an honest receiver in the local-permute-and-share step.

① The secret input from the receiver sent to $\mathcal{F}_{\mathsf{Input}}$ is of the form $\pi^*(\boldsymbol{x})$ for some $\pi^* \in \mathbf{S}_n$,
② $\pi^* = \pi$, where π is exact the permutation in the used AST $\langle\!\langle \pi \rangle\!\rangle$.

To ensure condition ①, it is sufficient to check that the shared ALSS vector from $\mathcal{F}_{\mathsf{Input}}$ is indeed a valid permutation of $\boldsymbol{\delta}$. We resort to an ideal functionality $\mathcal{F}_{\mathsf{PermCheck}}$ in Fig. 5 for the permutation check. Our first challenge is how to realize the permutation check in an efficiency-preserving way. Moreover, solely ensuring condition ① is not sufficient to restrict the receiver's behaviors. The receiver may provide a different permutation $\pi^* \neq \pi$ in its local permutation. This corresponds to that $\mathtt{R}$ reshares a vector $\pi^*(\boldsymbol{x}) = \pi(\boldsymbol{x}) + \boldsymbol{e}$ with error $\boldsymbol{e} = \pi^*(\boldsymbol{x}) - \pi(\boldsymbol{x})$, which breaks correctness trivially. So our second challenge is to ensure condition ② in an efficiency-preserving way. We show how to solve these challenges with new techniques and optimizations, followed by formal security analysis and proof.

Functionality $\mathcal{F}_{\mathsf{PermCheck}}$

Parameters: $n \in \mathbb{N}$; a dictionary Val storing all ALSS secrets.
Functionality: upon receiving $(\mathsf{PermCheck}, \{\mathsf{id}_i\}_{i\in[n]}, \{\mathsf{id}'_i\}_{i\in[n]})$ from $\mathtt{R}$ and $\mathtt{S}$:

- Fetch the vector $\boldsymbol{x}$ and $\boldsymbol{y}$ such that $\boldsymbol{x}_i = \mathsf{Val}[\mathsf{id}_i]$ and $\boldsymbol{y}_i = \mathsf{Val}[\mathsf{id}'_i]$ for $i \in [n]$.
- Check if $\boldsymbol{y}$ and $\boldsymbol{x}$ form a permutation relation. If yes, return True; otherwise, return False.

Fig. 5. The ideal permutation check functionality.

4.2 Ensuring ①: Round-efficient PermCheck

To securely realize the ideal permutation-check functionality, we start from a classic polynomial-based method [39,49]. Specifically, let

$$f_{\boldsymbol{u}}(X) = \prod_{i\in[n]} (X - u_i)$$

be a polynomial defined for a vector $\boldsymbol{u} \in \mathbb{F}^n$. For any $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}^n$, we have $f_{\boldsymbol{a}}(X) = f_{\boldsymbol{b}}(X)$ if and only if $\boldsymbol{a}$ and $\boldsymbol{b}$ satisfy a permutation relation. To check $f_{\boldsymbol{a}}(X) = f_{\boldsymbol{b}}(X)$, it is sufficient to check $f_{\boldsymbol{a}}(r) = f_{\boldsymbol{b}}(r)$ over a randomly picked $r \in \mathbb{F}$. This holds except with soundness error $n/|\mathbb{F}|$ following the Schwartz-Zippel Lemma [52,57]. We have $n/|\mathbb{F}|$ negligible in λ for a sufficiently large $\mathbb{F}$.

Figure 6 presents the permutation check protocol. It requires the parties to generate $\langle r \rangle$ for a uniformly random secret r via $\mathcal{F}_{\mathsf{Rand}}^{\langle\cdot\rangle}$, evaluate $f_{\boldsymbol{x}}(r)$ and $f_{\boldsymbol{y}}(r)$,

Protocol $\Pi_{\mathsf{PermCheck}}$

Parameter: Polynomial $f_{\boldsymbol{u}}(X) = \prod_{i\in[n]}(X - u_i)$ for $\boldsymbol{u} \in \mathbb{F}^n$.

Protocol: On receiving $\langle \boldsymbol{x}\rangle$ and $\langle \boldsymbol{y}\rangle$ for $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^n$:

1. $\langle r\rangle \leftarrow \mathcal{F}_{\mathsf{Rand}}^{\langle\cdot\rangle}(\mathbb{F})$, $\langle s\rangle \leftarrow \mathcal{F}_{\mathsf{Rand}}^{\langle\cdot\rangle}(\mathbb{F})$.
2. $\langle p\rangle \leftarrow (\langle x_0\rangle - \langle r\rangle)\cdot(\langle x_1\rangle - \langle r\rangle)\cdots(\langle x_{n-1}\rangle - \langle r\rangle)$. ▷ via $\Pi_{n\text{-mul}}$
3. $\langle q\rangle \leftarrow (\langle y_0\rangle - \langle r\rangle)\cdot(\langle y_1\rangle - \langle r\rangle)\cdots(\langle y_{n-1}\rangle - \langle r\rangle)$. ▷ via $\Pi_{n\text{-mul}}$
4. $\langle d\rangle \leftarrow \langle s\rangle \cdot (\langle p\rangle - \langle q\rangle)$. ▷ via $\mathcal{F}_{\mathsf{Mul}}^{\langle\cdot\rangle}$
5. $d \leftarrow \mathcal{F}_{\mathsf{OpenCheck}}^{\langle\cdot\rangle}(\langle d\rangle)$. If the call aborts or $d \neq 0$, return False. Otherwise, return True.

Fig. 6. The permutation check protocol.

and check whether $f_{\boldsymbol{x}}(r) - f_{\boldsymbol{y}}(r) = 0$. To protect against possible leakage when $f_{\boldsymbol{x}}(r) \neq f_{\boldsymbol{y}}(r)$, the parties compute $\langle d\rangle \leftarrow \langle s\rangle \cdot (\langle p\rangle - \langle q\rangle)$, where s is the secret to mask $f_{\boldsymbol{x}}(r) - f_{\boldsymbol{y}}(r)$. The parties then open and check $d = 0$.

The permutation check protocol is designed in the ABB fashion, where honest behaviors can be enforced via existing SPDZ malicious-security functionalities. However, it requires n-input secret-shared multiplication to perform polynomial evalaution $\langle f_{\boldsymbol{x}}(r)\rangle$ and $\langle f_{\boldsymbol{y}}(r)\rangle$. While polynomial evaluation can be realized using $O(n)$ MTs with linear online communication, it still requires $O(\log n)$ rounds following a tree-structure multiplication evaluation process. If we directly evaluate this permutation check protocol, we cannot reach our efficiency goal.

Round-Efficient n-Input Multiplication. To obtain the expected efficiency property, we propose an n-input multiplication protocol with linear and constant round online communication. Our method is inspired by the "mask-then-inverse" trick [18,56], but it requires additional cares on security, which we will analyze formally. To start with, suppose the parties pre-generate $n + 1$ random secrets

$$\langle s\rangle, \langle \boldsymbol{r}\rangle = (\langle r_0\rangle, \langle r_1\rangle, \cdots, \langle r_{n-2}\rangle, \langle r_{n-1}\rangle)$$

in the offline phase satisfying $s = \prod_{i\in[n]} r_i^{-1}$. The parties can securely compute n-input multiplication $\langle y\rangle = \prod_{i\in[n]}\langle x_i\rangle$. Specifically, the parties compute

$$\langle m_i\rangle \leftarrow \langle x_i\rangle \cdot \langle r_i\rangle,$$

and securely open m_i for all $i \in [n]$ in parallel with linear and one-round communication. Then the parties compute

$$\langle y\rangle \leftarrow (\prod_{i\in[n]} m_i)\cdot\langle s\rangle,$$

which only requires local computation. Correctness holds since $\{r_i\}_{i\in[n]}$ all cancel out during the computation:

$$\langle y\rangle = (\prod_{i\in[n]} m_i)\cdot\langle s\rangle = (\prod_{i\in[n]}(x_i\cdot r_i))\cdot\langle\prod_{i\in[n]} r_i^{-1}\rangle = \langle\prod_{i\in[n]} x_i\rangle.$$

To generate $\langle s\rangle = \langle \prod_{i\in[n]} r_i^{-1}\rangle$ from $(\langle r_0\rangle, \langle r_1\rangle, \ldots, \langle r_{n-1}\rangle)$ in the preprocessing phase, the parties generate a random ALSS secret $\langle\omega\rangle$ and compute

$$\langle\beta\rangle \leftarrow \langle\omega\rangle \cdot \langle r_0\rangle \cdot \langle r_1\rangle \cdots \langle r_{n-1}\rangle,$$

The parties then open β and compute

$$\langle s\rangle \leftarrow \beta^{-1} \cdot \langle\omega\rangle = \omega^{-1} \cdot \prod_{i\in[n]} r_i^{-1} \cdot \langle\omega\rangle = \langle \prod_{i\in[n]} r_i^{-1}\rangle,$$

where ω serves as a mask to hide $\prod_{i\in[n]} r_i^{-1}$ when opening β. Here we require none of $\omega, r_0, r_1, \ldots, r_{n-1}$ be zero to compute the inverse β^{-1}; this holds except with probability at most $(n+1)/|\mathbb{F}|$, which is negligible for sufficiently large $\mathbb{F}$.

Figure 7 presents the n-input multiplication protocol. The online phase requires n multiplications and n open operations with linear cost and constant-round complexity. Plugging $\Pi_{n\text{-mul}}$ protocol for n-input multiplication, $\Pi_{\mathsf{PermCheck}}$ achieves linear cost and constant-round online communication, preserving the desired efficiency property.

Protocol $\Pi_{n\text{-mul}}$

[**Offline**] On input n:

1. $(\langle r_0\rangle, \langle r_1\rangle, \cdots, \langle r_{n-1}\rangle, \langle\omega\rangle) \leftarrow \mathcal{F}_{\mathsf{Rand}}^{\langle\cdot\rangle}(\mathbb{F}^{n+1})$.
2. $\langle\beta\rangle \leftarrow \langle\omega\rangle \cdot \langle r_0\rangle \cdot \langle r_1\rangle \cdots \langle r_{n-1}\rangle$. ▷ via $\mathcal{F}_{\mathsf{Mul}}^{\langle\cdot\rangle}$
3. $\beta \leftarrow \mathcal{F}_{\mathsf{OpenCheck}}^{\langle\cdot\rangle}(\langle\beta\rangle)$. Abort if open or check fails.
4. $\langle s\rangle \leftarrow \beta^{-1} \cdot \langle\omega\rangle$. ▷ via $\mathcal{F}_{\mathsf{Mul}}^{\langle\cdot\rangle}$
5. Return $\langle \boldsymbol{r}\rangle$, $\langle s\rangle$.

[**Online**] On input sharings $\langle x_0\rangle, \langle x_1\rangle, \cdots, \langle x_{n-1}\rangle$:

1. $\langle m_i\rangle \leftarrow \langle x_i\rangle \cdot \langle r_i\rangle$ for $i \in [n]$. ▷ via $\mathcal{F}_{\mathsf{Mul}}^{\langle\cdot\rangle}$
2. $m_i \leftarrow \mathcal{F}_{\mathsf{OpenCheck}}(\langle m_i\rangle)$ for $i \in [n]$, $m \leftarrow \prod_{i\in[n]} m_i$.
3. $\langle y\rangle \leftarrow m \cdot \langle s\rangle$.

Fig. 7. The n-input multiplication protocol $\Pi_{n\text{-mul}}$.

Remark 1. One might expect that $\Pi_{n\text{-mul}}$ securely realizes some ideal n-input multiplication functionality. We find that this is impossible for arbitrary input distributions of $\boldsymbol{x}$ as messages m_i may leak information about x_i under certain distributions. To understand this intuitively: $m_i = 0$ if and only if $x_i = 0$ or $r_i = 0$. As r_i is sampled uniformly from $\mathbb{F}$, the probability that $r_i = 0$ is only $1/|\mathbb{F}|$. Thus, the event $m_i = 0$ is overwhelmingly determined by $x_i = 0$. When $x_i = 0$ occurs with non-negligible probability, observing $m_i = 0$ inevitably reveals information about $x_i = 0$, except with probability $1/|\mathbb{F}|$ when $r_i = 0$. Nevertheless, we will show that such a bad event only happens with negligible probability when using $\Pi_{n\text{-mul}}$ in our case.

4.3 Security Proof

We prove that $\Pi_{\mathsf{PermCheck}}$ that relies on the n-input multiplication protocol $\Pi_{n\text{-mul}}$ securely realizes $\mathcal{F}_{\mathsf{PermCheck}}$, as shown by Theorem 1.

Theorem 1. *$\Pi_{\mathsf{PermCheck}}$ protocol using $\Pi_{n\text{-mul}}$ securely realizes $\mathcal{F}_{\mathsf{PermCheck}}$ in the $\mathcal{F}^{\langle\cdot\rangle}$-hybrid model with statistical error $(4n+1)/|\mathbb{F}|$.*

Proof. Let $\mathcal{A}$ be any PPT adversary who statically corrupts one party. We construct a PPT simulator $\mathcal{S}$ that simulates the adversary's view. When $\mathcal{A}$ aborts or $\mathcal{S}$ terminates the simulation, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs. Note that in $\Pi_{\mathsf{PermCheck}}$, the only revealed information is from β and $\{m_i\}_{i\in[n]}$ in $\Pi_{n\text{-mul}}$, and all other operations are ABB calls to the ideal commands in $\mathcal{F}^{\langle\cdot\rangle}$. In the hybrid model, the simulator $\mathcal{S}$ works as follows:

1. To simulate the opened value β in the offline phase of $\Pi_{n\text{-mul}}$, $\mathcal{S}$ simply samples $\beta \xleftarrow{\$} \mathbb{F} \setminus \{0\}$ and sends β to $\mathcal{A}$.
2. To simulate the opened values $\{m_i\}_{i\in[n]}$ in the online phase of $\Pi_{n\text{-mul}}$, $\mathcal{S}$ simply sampels $m_i \xleftarrow{\$} \mathbb{F}$ for $i \in [n]$ and sends $\{m_i\}_{i\in[n]}$ to $\mathcal{A}$.
3. Note that all other operations in $\Pi_{n\text{-mul}}$ and $\Pi_{\mathsf{PermCheck}}$ are calls to $\mathcal{F}^{\langle\cdot\rangle}$. For any ideal command call from $\mathcal{A}$ to $\mathcal{F}^{\langle\cdot\rangle}_{\mathsf{Mul}}$, $\mathcal{F}^{\langle\cdot\rangle}_{\mathsf{OpenCheck}}$, $\mathcal{F}^{\langle\cdot\rangle}_{\mathsf{Rand}}$, $\mathcal{S}$ aborts if $\mathcal{A}$ adds any error to any command call.
4. If $\mathcal{A}$ does not abort in any step, $\mathcal{S}$ sends $(\mathsf{PermCheck}, \{\mathsf{id}_i\}_{i\in[n]}, \{\mathsf{id}'_i\}_{i\in[n]})$ to the ideal $\mathcal{F}_{\mathsf{PermCheck}}$ functionality, where $\{\mathsf{id}_i\}_{i\in[n]}, \{\mathsf{id}'_i\}_{i\in[n]}$ are the indexes of two ALSS vectors being checked. $\mathcal{S}$ aborts if $\mathcal{F}_{\mathsf{PermCheck}}$ return False.
5. $\mathcal{S}$ outputs what $\mathcal{A}$ outputs.

We prove that the ideal-world execution is indistinguishable from real-protoocl execution via a sequence of hybrid games.

G_0: G_0 is exactly the real-protocol execution.

G_1:G_1 differs from G_0 in that G_0 cannot proceed if $\beta = \omega \cdot r_0 \cdot r_1 \cdots r_{n-1}$ happens to be 0. G_1 never aborts at the same step since $\beta \xleftarrow{\$} \mathbb{F} \setminus \{0\}$. The abort probability between two games is statistically indistinguishable for sufficiently large $\mathbb{F}$. To see this, each of ω, $r_0, r_1 \cdots r_{n-1}$ is uniformaly sampled over $\mathbb{F}$ (in the $\mathcal{F}^{\langle\cdot\rangle}_{\mathsf{Rand}}$-hybrid model), the probability that $\beta = 0$ is bounded by $(n+1)/|\mathbb{F}|$ following the union bound.

G_2: G_2 differes from G_1 in that when computing $\langle p\rangle$ using $\Pi_{n\text{-mul}}$, G_2 samples $m_i \xleftarrow{\$} \mathbb{F}$ for all $i \in [n]$, while in G_1 we have $m_i = x_i \cdot r_i$ where $r_i \xleftarrow{\$} \mathbb{F}$ and x_i is from the input of $\Pi_{n\text{-mul}}$. We need to prove that the distribution of $m_i = x_i \cdot r_i$ in G_1 is indistinguishable from the distribution of m_i in G_2 for all $i \in [n]$.

To this end, we define a sequence of hybrid games $\mathsf{G}_{1,i}$ for $i \in [n+1]$, where $\mathsf{G}_{1,i}$ is a game such that for $j < i$, m_i is generated exactly as in G_2, while m_j for $j \geq i$ is generated as in G_1. Clearly, $\mathsf{G}_{1,0} = \mathsf{G}_1$ and $\mathsf{G}_{1,n} = \mathsf{G}_2$. Now we show that $\mathsf{G}_{1,i}$ is indistinguishable from $\mathsf{G}_{1,i+1}$ for $i \in [n]$. To prove this, we introduce Lemma 1. Lemma 1 shows that for any random variable X following any distribution and any $R \sim \mathsf{Uniform}(\mathbb{F})$, the statistical distance between the distributions of $X \cdot R$ and the uniform distribution is bounded by $\Pr[X = 0]$.

Lemma 1. *Let X be a random variable following some (unknown) distribution over $\mathbb{F}$ with $\Pr[X=0]=p$ for $0\le p\le 1$, $R\sim \mathsf{Uniform}(\mathbb{F})$ and $U\sim \mathsf{Uniform}(\mathbb{F})$ follow uniform distribution over $\mathbb{F}$, and the random variable $M=X\cdot R$, then $\mathsf{SD}(M,U)\le p$.*

Proof. We discuss $\Pr[M=m]$ depending on two cases: $m=0$ or $m\neq 0$. For $m=0$, $\Pr[M=0]=\Pr[X=0]+\Pr[X\neq 0\wedge R=0]=\Pr[X=0]+\Pr[X\neq 0]\cdot\Pr[R=0]=p+(1-p)/|\mathbb{F}|$. For $m\neq 0$, $\Pr[M=m]=\Pr[X\cdot R=m]=\Pr[X\neq 0\wedge R=m\cdot X^{-1}]=\sum_{x\in\mathbb{F}\setminus\{0\}}\Pr[X=x\wedge R=m\cdot x^{-1}]=\sum_{x\in\mathbb{F}\setminus\{0\}}\Pr[X=x]\cdot\Pr[R=m\cdot x^{-1}]=(1-p)/|\mathbb{F}|$. Overall, we have:

$$\Pr[M=m]=\begin{cases}p+\frac{1-p}{|\mathbb{F}|}, & m=0,\\ \frac{1-p}{|\mathbb{F}|}, & m\neq 0.\end{cases} \tag{2}$$

Since U follows the uniform distribution over $\mathbb{F}$, we have $\Pr[U=m]=1/|\mathbb{F}|$ for $m\in\mathbb{F}$. By Eq. (2) and the defintion of statistical distance, we have

$$\begin{aligned}\mathsf{SD}(M,U)=&\frac{1}{2}\sum_{m\in\mathbb{F}}|\Pr[M=m]-\Pr[U=m]|\\ =&\frac{1}{2}\Big(\Big|\Pr[M=0]-\frac{1}{|\mathbb{F}|}\Big|+\sum_{m\in\mathbb{F}\setminus\{0\}}\Big|\Pr[M=m]-\frac{1}{|\mathbb{F}|}\Big|\Big)\\ =&\frac{1}{2}(p+\frac{1-p}{|\mathbb{F}|}-\frac{1}{|\mathbb{F}|})+\frac{1}{2}(\mathbb{F}-1)\cdot\Big|\frac{1-p}{|\mathbb{F}|}-\frac{1}{|\mathbb{F}|}\Big|\\ =&p\cdot(1-\frac{1}{|\mathbb{F}|})\le p,\end{aligned}$$

where $p\cdot(1-\frac{1}{|\mathbb{F}|})=p$ holds if $p=0$.

When computing $p=f_{\boldsymbol{x}}(r)=\prod_{i\in[n]}(x_i-r)$, each term x_i-r corresponds to the input of $\Pi_{n\text{-}\mathsf{mul}}$. Following Lemma 1, we expect the probability that $x_i-r=0$ to be negligible in λ. If this holds, the distribution of $(x_i-r)\cdot r_i$ is still statistically indistinguishable from the uniform distribution over $\mathbb{F}$. This actually holds: since r is uniformly distributed over $\mathbb{F}$, the probability that $x_i-r=0$ is negligible in λ for large enough $\mathbb{F}$. Hence, we have $\mathsf{G}_{1,i}$ is statistically indistinguishable from $\mathsf{G}_{1,i+1}$, with statistical error $1/|\mathbb{F}|$ for $i\in[n]$. Combining all intemidiate game $\mathsf{G}_{1,i}$ for $i\in[n]$, the statistical distinguishability between G_1 and G_2 is bounded by $n/|\mathbb{F}|$.

G_3: G_3 differes from G_2 in that when computing $\langle q\rangle$ using $\Pi_{n\text{-}\mathsf{mul}}$, G_3 samples $m_i\xleftarrow{\$}\mathbb{F}$ for all $i\in[n]$, while in G_2 we have $m_i=x_i\cdot r_i$ where $r_i\xleftarrow{\$}\mathbb{F}$ and x_i is from the input of $\Pi_{n\text{-}\mathsf{mul}}$. Following the same analysis as above, the statistical distinguishability between G_2 and G_3 is bounded by $n/|\mathbb{F}|$.

G_4: G_4 directly sends $(\mathsf{PermCheck},\{\mathsf{id}_i\}_{i\in[n]},\{\mathsf{id}'_i\}_{i\in[n]})$ to the ideal $\mathcal{F}_{\mathsf{PermCheck}}$ functionality, where $\{\mathsf{id}_i\}_{i\in[n]},\{\mathsf{id}'_i\}_{i\in[n]}$ are the indexes of two ALSS vectors to be checked. G_4 aborts if $\mathcal{F}_{\mathsf{PermCheck}}$ returns False. We note that $\mathcal{F}_{\mathsf{PermCheck}}$

always aborts if the two ALSS vectors do not satisfy a permutation relation, while in G_3, the permutation check is performed by evaluating permutation-check polynomials, with soundness error $n/|\mathbb{F}|$ following the Schwartz-Zippel Lemma as we analysed in Sect. 4.2. Thus, G_4 is indistinguishable from G_3 with statistical error $n/|\mathbb{F}|$. Note that G_4 is exactly the ideal-world execution.

Combining all the above games, G_0 is statistically indistinguishable from G_4, with statistical error $(4n+1)/|\mathbb{F}|$.

4.4 Ensuring ②: Two PermChecks Suffice

Permutation check over $\langle \boldsymbol{\delta} \rangle$ and $\langle \boldsymbol{c} \rangle$ ensures condition ①, but not condition ②. The receiver may provide π^* with $\pi^* \neq \pi$ that is not the expected permutation from the used AST $\langle\!\langle \pi \rangle\!\rangle$. In this case, the receiver can still pass the permutation check $\Pi_{\mathsf{PermCheck}}(\langle \boldsymbol{\delta} \rangle, \langle \boldsymbol{c} \rangle)$, but correctness cannot hold. Lemma 2 shows that it is sufficient to further check whether $\langle \boldsymbol{x} \rangle$ and $\langle \boldsymbol{y} \rangle$ satisfy a permutation relation.

Lemma 2. *Let $\langle\!\langle \pi \rangle\!\rangle$ be a non-leaky and well-formed AST. If $\mathcal{F}_{\mathsf{PermCheck}}(\langle \boldsymbol{\delta} \rangle, \langle \boldsymbol{c} \rangle)$ and $\mathcal{F}_{\mathsf{PermCheck}}(\langle \boldsymbol{x} \rangle, \langle \boldsymbol{y} \rangle)$ pass without abort, then we have both conditons ① and ② holds.*

Proof. $\mathcal{F}_{\mathsf{PermCheck}}(\langle \boldsymbol{\delta} \rangle, \langle \boldsymbol{c} \rangle)$ ensurs condition ①. We prove that $\mathcal{F}_{\mathsf{PermCheck}}(\langle \boldsymbol{x} \rangle, \langle \boldsymbol{y} \rangle)$ further ensures condition ②. To see this, suppose $\mathtt{R}$ share $\pi^*(\boldsymbol{c})$ for some $\pi^* \neq \pi$. We have

$$\langle \boldsymbol{y} \rangle = \langle \boldsymbol{c} \rangle + \langle \boldsymbol{b} \rangle = \langle \pi^*(\boldsymbol{x} - \boldsymbol{a}) \rangle + \langle \pi(\boldsymbol{a}) \rangle = \langle \pi^*(\boldsymbol{x}) \rangle + \langle \pi(\boldsymbol{a}) - \pi^*(\boldsymbol{a}) \rangle.$$

Let $\boldsymbol{e} = \pi(\boldsymbol{a}) - \pi^*(\boldsymbol{a})$, we prove that $\boldsymbol{e} \neq \boldsymbol{0}$ for any $\pi^* \neq \pi$. To see this, when $\pi^* \neq \pi$, there must exist at least $i \in [n]$ such that $\pi(i) \neq \pi^*(i)$. Since

$$\boldsymbol{e}_i = \pi(\boldsymbol{a})_i - \pi^*(\boldsymbol{a})_i = \boldsymbol{a}_{\pi^*(i)} - \boldsymbol{a}_{\pi(i)},$$

it follows $\boldsymbol{e}_i \neq 0$ holds except with probability $1/|\mathbb{F}|$ since $\boldsymbol{a}$ is uniformally randomly sampled from $\mathbb{F}^n$. If $\boldsymbol{e} \neq \boldsymbol{0}$, $\mathcal{F}_{\mathsf{PermCheck}}(\langle \boldsymbol{x} \rangle, \langle \boldsymbol{y} \rangle)$ will detect the error.

4.5 The Proposed OSS Protocol

Figure 8 shows the final OSS protocol $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$ in the $(\mathcal{F}^{\langle\cdot\rangle}, \mathcal{F}_{\mathsf{GenAST}}, \mathcal{F}_{\mathsf{PermCheck}})$-hybrid model. Security of $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$ follows from the above analysis naturally.

One may wonder whether we can omit the check $\mathcal{F}_{\mathsf{PermCheck}}(\langle \boldsymbol{\delta} \rangle, \langle \boldsymbol{c} \rangle)$ and solely conduct $\mathcal{F}_{\mathsf{PermCheck}}(\langle \boldsymbol{x} \rangle, \langle \boldsymbol{y} \rangle)$. We stress that $\mathcal{F}_{\mathsf{PermCheck}}(\langle \boldsymbol{x} \rangle, \langle \boldsymbol{y} \rangle)$ alone is insufficient for security: it merely guarantees that $\langle \boldsymbol{x} \rangle$ and $\langle \boldsymbol{y} \rangle$ satisfy a permutation relation under *some* permutation, which enforces neither condition ① nor ②. Furthermore, we show a concrete selective-failure attack from $\mathtt{R}$ if $\mathcal{F}_{\mathsf{PermCheck}}(\langle \boldsymbol{\delta} \rangle, \langle \boldsymbol{c} \rangle)$ is omitted. This attack allows $\mathtt{R}$ to guess the difference between x_i and x_j for any $i, j \in [n]$. We sketch the attack as follows:

Protocol $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$

[**Shuffle**] On receiving an ALSS vector $\langle \boldsymbol{x} \rangle$ and an AST correlation $\langle\!\langle \pi \rangle\!\rangle = (\pi, \langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle)$:

1. $\langle \boldsymbol{\delta} \rangle \leftarrow \langle \boldsymbol{x} \rangle - \langle \boldsymbol{a} \rangle$.
2. $\boldsymbol{\delta} \leftarrow \mathcal{F}_{\mathsf{OpenCheck}}^{\langle \cdot \rangle}(\langle \boldsymbol{\delta} \rangle)$. Abort if open or check fails.
3. $\mathtt{R}$ computes $\boldsymbol{c} \leftarrow \pi(\boldsymbol{\delta})$ and shares $\langle \boldsymbol{c} \rangle$ between the parties via $\mathcal{F}_{\mathsf{Input}}^{\langle \cdot \rangle}$.
4. $\langle \boldsymbol{y} \rangle \leftarrow \langle \boldsymbol{c} \rangle + \langle \boldsymbol{b} \rangle$.
5. Run $\mathcal{F}_{\mathsf{PermCheck}}(\langle \boldsymbol{\delta} \rangle, \langle \boldsymbol{c} \rangle)$ and $\mathcal{F}_{\mathsf{PermCheck}}(\langle \boldsymbol{x} \rangle, \langle \boldsymbol{y} \rangle)$. If the checks fail, abort.
6. Return $\langle \boldsymbol{y} \rangle$.

Fig. 8. The AST-based OSS protocol.

- After obtaining the correct opened vector $\boldsymbol{\delta}$, $\mathtt{R}$ guesses $d = x_i - x_j$, $\mathtt{R}$ injects an error vector $\boldsymbol{e}$ such that

$$e_i = -d, e_j = d, \text{ and } e_k = 0 \text{ for any } k \in [n] \setminus \{i, j\}$$

- Insteaf of sharing $\pi(\boldsymbol{\delta})$ via $\mathcal{F}_{\mathsf{Input}}$, $\mathtt{R}$ shares $\boldsymbol{c} = \pi(\boldsymbol{\delta} + \boldsymbol{e})$ between the parties.

By definition, the parties share $\langle \boldsymbol{y} \rangle = \langle \pi(\boldsymbol{x} + \boldsymbol{e}) \rangle$. If $\mathtt{R}$ guessed $d = x_i - x_j$ correctly, then $\boldsymbol{x} + \boldsymbol{e}$ is still a permutation of $\boldsymbol{x}$. In this case, x_i and x_j are swapped due to the injected errors; hence, $\boldsymbol{y}$ is still a permutation of $\boldsymbol{x}$, but the permutation is not π. As a result, $\mathcal{F}_{\mathsf{PermCheck}}(\langle \boldsymbol{x} \rangle, \langle \boldsymbol{y} \rangle)$ still passes, yet $\mathtt{R}$ learns $d = x_i - x_j$ without being caught. This concrete attack highlights why we cannot skip the permutation check $\mathcal{F}_{\mathsf{PermCheck}}(\langle \boldsymbol{\delta} \rangle, \langle \boldsymbol{c} \rangle)$.

Security. Theorem 2 shows that $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$ securely computes functionality $\mathcal{F}_{\mathsf{oss}}^{[\mathtt{R}]}$ in the $(\mathcal{F}^{\langle \cdot \rangle}, \mathcal{F}_{\mathsf{GenAST}}, \mathcal{F}_{\mathsf{PermCheck}})$-hybrid model. Security of $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$ follows from security of each ideal command/functionality, since $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$ only calls avaiable ideal commands that we have provided sufficient proof and analysis in Sect. 4.3 and Lemma 2.

Theorem 2. *$\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$ securely realizes $\mathcal{F}_{\mathsf{oss}}^{[\mathtt{R}]}$ in the $(\mathcal{F}^{\langle \cdot \rangle}, \mathcal{F}_{\mathsf{GenAST}}, \mathcal{F}_{\mathsf{PermCheck}})$-hybrid model.*

5 MOSAC via Layer-Two Authentication

5.1 High-Level Idea

The prior MOSAC construction leverages permutation checks to enforce that the receiver provides $\pi(\boldsymbol{\delta})$ back to the ALSS world, requiring relatively expensive n-input multiplication evaluation. In this section, we propose a simpler consistency check to avoid evaluating permutation-check polynomials.

Our new consistency check leverages an additional layer of authentication to detect possible errors from $\mathtt{R}$ in the resharing phase, which we call *layer-two authentication*. Our idea is that we authenticate $\langle \boldsymbol{x} \rangle$ *before* shuffling via another

layer of SPDZ-like MAC. When opening $\boldsymbol{\delta}$, the original layer of SPDZ MAC ensures the correct opening of δ, while the new layer of authentication remains to enforce consistency when combined with AST, which is how the name *layer-two authentication* comes from.

Protocol $\Pi^{[\mathtt{R}]}_{\mathsf{L2\text{-}oss}}$

[**Shuffle**] On receiving $\langle \boldsymbol{x} \rangle$ and AST $\langle\!\langle \pi \rangle\!\rangle = (\pi, \langle \boldsymbol{a}^{(0)} \rangle, \langle \boldsymbol{a}^{(1)} \rangle, \langle \boldsymbol{b}^{(0)} \rangle, \langle \boldsymbol{b}^{(1)} \rangle)$:

1. Generate a random secret layer-two MAC key sharing $\langle \psi \rangle \leftarrow \mathcal{F}^{\langle \cdot \rangle}_{\mathsf{Rand}}(\mathbb{F})$.
2. Compute layer-two MAC $\langle \psi \cdot \boldsymbol{x} \rangle \leftarrow \langle \psi \rangle \cdot \langle \boldsymbol{x} \rangle$ via $\mathcal{F}^{\langle \cdot \rangle}_{\mathsf{Mul}}$.
3. Compute $\langle \boldsymbol{\delta}^{(0)} \rangle \leftarrow \langle \boldsymbol{x} \rangle - \langle \boldsymbol{a}^{(0)} \rangle, \langle \boldsymbol{\delta}^{(1)} \rangle \leftarrow \langle \psi \cdot \boldsymbol{x} \rangle - \langle \boldsymbol{a}^{(1)} \rangle$.
4. Securely open $\boldsymbol{\delta}^{(0)} \leftarrow \mathcal{F}^{\langle \cdot \rangle}_{\mathsf{OpenCheck}}(\langle \boldsymbol{\delta}^{(0)} \rangle)$ and $\boldsymbol{\delta}^{(1)} \leftarrow \mathcal{F}^{\langle \cdot \rangle}_{\mathsf{OpenCheck}}(\langle \boldsymbol{\delta}^{(1)} \rangle)$. Abort if open or check fails.
5. $\mathtt{R}$ computes $\boldsymbol{c}^{(0)} \leftarrow \pi(\boldsymbol{\delta}^{(\mathbf{0})})$, $\boldsymbol{c}^{(1)} \leftarrow \pi(\boldsymbol{\delta}^{(\mathbf{1})})$, and shares $\langle \boldsymbol{c}^{(0)} \rangle$ and $\langle \boldsymbol{c}^{(1)} \rangle$ via $\mathcal{F}^{\langle \cdot \rangle}_{\mathsf{Input}}$.
6. The parties compute $\langle \boldsymbol{y}^{(0)} \rangle \leftarrow \langle \boldsymbol{c}^{(0)} \rangle + \langle \boldsymbol{b}^{(0)} \rangle$ and $\langle \boldsymbol{y}^{(1)} \rangle \leftarrow \langle \boldsymbol{c}^{(1)} \rangle + \langle \boldsymbol{b}^{(1)} \rangle$.
7. $b \leftarrow \Pi_{\mathsf{L2MacCheck}}(\langle \psi \rangle, \langle \boldsymbol{y}^{(0)} \rangle, \langle \boldsymbol{y}^{(1)} \rangle)$. If $b = \text{False}$, abort.
8. Return $\langle \boldsymbol{y}^{(0)} \rangle$.

Fig. 9. MOSAC shuffle from layer-two authentication.

5.2 The Proposed OSS Protocol

Figure 9 shows the formal protocol specification. Specifically, the parites additionally samples a secret-shared layer-two MAC key ψ and compute layer-two MAC $\langle \psi \cdot \boldsymbol{x} \rangle$. To shuffle two columns of ALSS vectors, we use an AST with $\mathbb{D} = \mathbb{F}^2$ that supports a two-column shuffle. Such a two-column AST is defined as

$$\langle\!\langle \pi \rangle\!\rangle = (\pi, \langle \boldsymbol{a}^{(0)} \rangle, \langle \boldsymbol{a}^{(1)} \rangle, \langle \boldsymbol{b}^{(0)} \rangle, \langle \boldsymbol{b}^{(1)} \rangle),$$

where $\boldsymbol{b}^{(i)} = \pi(\boldsymbol{a}^{(i)})$ for $i \in \{0, 1\}$. To shuffle two column of ALSS vectors $(\langle \boldsymbol{x} \rangle, \langle \psi \cdot \boldsymbol{x} \rangle)$, the parties comptue

$$\langle \boldsymbol{\delta}^{(0)} \rangle \leftarrow \langle \boldsymbol{x} \rangle - \langle \boldsymbol{a}^{(0)} \rangle, \langle \boldsymbol{\delta}^{(1)} \rangle \leftarrow \langle \psi \cdot \boldsymbol{x} \rangle - \langle \boldsymbol{a}^{(1)} \rangle,$$

and *correctly* open $\langle \boldsymbol{\delta}^{(0)} \rangle$ and $\langle \boldsymbol{\delta}^{(1)} \rangle$ to obtain $\boldsymbol{\delta}^{(0)}$ and $\boldsymbol{\delta}^{(1)}$. Note that $\boldsymbol{\delta}^{(0)}$ and $\boldsymbol{\delta}^{(1)}$ perfectly hide $\boldsymbol{x}$ and $\psi \cdot \boldsymbol{x}$ since $\boldsymbol{a}^{(0)}$ and $\boldsymbol{a}^{(1)}$ are uniformally sampled from $\mathbb{F}^n$. Then the receiver $\mathtt{R}$ computes

$$\boldsymbol{c}^{(0)} \leftarrow \pi(\boldsymbol{\delta}^{(0)}), \boldsymbol{c}^{(1)} \leftarrow \pi(\boldsymbol{\delta}^{(1)}),$$

and share $(\boldsymbol{c}^{(0)}, \boldsymbol{c}^{(1)})$ back to the ALSS world via $\mathcal{F}^{\langle \cdot \rangle}_{\mathsf{Input}}$. The parties comptutes

$$\langle \boldsymbol{y}^{(0)} \rangle \leftarrow \langle \boldsymbol{c}^{(0)} \rangle + \langle \boldsymbol{b}^{(0)} \rangle, \langle \boldsymbol{y}^{(1)} \rangle \leftarrow \langle \boldsymbol{c}^{(1)} \rangle + \langle \boldsymbol{b}^{(1)} \rangle.$$

We can check $\langle \boldsymbol{y}^{(0)} \rangle = \langle \pi(\boldsymbol{x}) \rangle$ and $\langle \boldsymbol{y}^{(1)} \rangle = \langle \psi \cdot \pi(\boldsymbol{x}) \rangle$ hold if the parties follow the protocol faithfully. To detect errors from the permutate-and-share step, the parties run a layer-two MAC check protocol from Fig. 10, which is adapted from the well-known batch MAC check protocols from existing SPDZ-family protocols [19,21,36].

Protocol $\Pi_{\mathsf{L2MacCheck}}$

Protocol: On receiving the layer-two MAC key ψ, $\langle \boldsymbol{y}^{(0)} \rangle$, and $\langle \boldsymbol{y}^{(1)} \rangle$, check whether $\boldsymbol{y}^{(0)} = \psi \cdot \boldsymbol{y}^{(1)}$:

1. Generate $\langle r \rangle \leftarrow \mathcal{F}_{\mathsf{Rand}}^{\langle \cdot \rangle}(\mathbb{F})$. Compute $\langle \psi \cdot r \rangle \leftarrow \langle \psi \rangle \cdot \langle r \rangle$
2. $(\chi_1, \chi_2, \cdots, \chi_n) \leftarrow \mathcal{F}_{\mathsf{Coin}}(\mathbb{F}^n)$. Compute $\langle m^{(0)} \rangle \leftarrow \langle r \rangle + \sum_{i \in [n]} \chi_i \cdot \langle y_i^{(0)} \rangle$ and $\langle m^{(1)} \rangle \leftarrow \langle \psi \cdot r \rangle + \sum_{i \in [n]} \chi_i \cdot \langle y_i^{(1)} \rangle$.
3. Open $m^{(0)} \leftarrow \mathcal{F}_{\mathsf{OpenCheck}}^{\langle \cdot \rangle}(\langle m^{(0)} \rangle)$.
4. Compute $\langle o \rangle \leftarrow m^{(0)} \cdot \langle \psi \rangle - \langle m^{(1)} \rangle$. Open $o \leftarrow \mathcal{F}_{\mathsf{OpenCheck}}^{\langle \cdot \rangle}(\langle o \rangle)$.
5. If $o \neq 0$ or the protocol aborts, return False; Otherwise, return True.

Fig. 10. The layer-2 MAC check protocol.

The OSS protocol from layer-two authentication avoids permutation checks. Instead, it requires the parties to generate a layer-two MAC via secure multiplication between a single layer-two MAC key and an ALSS vector, and then the ALSS vector, together with its layer-two MAC vector, is permuted by a two-dimensional AST. Overall, the computation and communication complexities are still linear, and the round complexity remains constant.

Security. Here we sketch its security intuition. Recall that the malicious R may inject arbratray errors into $\pi(\boldsymbol{\delta}^{(0)})$ and/or $\pi(\boldsymbol{\delta}^{(1)})$. We rely on the layer-two MAC check and the well-formedness of the AST to detect errors (step 7, Fig. 9). We show that if R introduces any error in the sharing phase, the layer-two MAC can always detect the error, except with negligible probability. Let $\tilde{\boldsymbol{c}}^{(0)}$ and $\tilde{\boldsymbol{c}}^{(1)}$ be the acutal input from R, and let $\boldsymbol{e}^{(0)} \leftarrow \tilde{\boldsymbol{c}}^{(0)} - \boldsymbol{c}^{(0)}$ and $\boldsymbol{e}^{(1)} \leftarrow \tilde{\boldsymbol{c}}^{(1)} - \boldsymbol{c}^{(1)}$ be the introduced error vectors. Then we have

$$\langle \boldsymbol{y}^{(0)} \rangle = \langle \tilde{\boldsymbol{c}}^{(0)} \rangle + \langle \boldsymbol{b}^{(0)} \rangle = \langle \boldsymbol{c}^{(0)} \rangle + \langle \boldsymbol{b}^{(0)} \rangle + \langle \boldsymbol{e}^{(0)} \rangle = \langle \pi(\boldsymbol{x}) \rangle + \langle \boldsymbol{e}^{(0)} \rangle,$$
$$\langle \boldsymbol{y}^{(1)} \rangle = \langle \tilde{\boldsymbol{c}}^{(1)} \rangle + \langle \boldsymbol{b}^{(1)} \rangle = \langle \boldsymbol{c}^{(1)} \rangle + \langle \boldsymbol{b}^{(1)} \rangle + \langle \boldsymbol{e}^{(1)} \rangle = \langle \psi \cdot \pi(\boldsymbol{x}) \rangle + \langle \boldsymbol{e}^{(1)} \rangle.$$

To pass the layer-two MAC check, it requires that

$$\psi \cdot (\pi(\boldsymbol{x}) + \boldsymbol{e}^{(0)} + r) = \psi \cdot (\pi(\boldsymbol{x}) + r) + \boldsymbol{e}^{(1)} \Rightarrow \psi \cdot \boldsymbol{e}^{(0)} = \boldsymbol{e}^{(1)}.$$

If R introduces an error and passes the layer-two MAC check, it equaliently means that R can recover the layer-two MAC key ψ from $\boldsymbol{e}^{(1)}$ and $\boldsymbol{e}^{(0)}$. But this only happens with negligible probability. Thus, we conclude that layer-two MAC is sufficient to detect the errors from R.

Theorem 3 formalizes the security of $\Pi^{[\mathbb{R}]}_{\mathsf{L2\text{-}oss}}$ using the layer-two authentication. We will provide a formal proof in the full version.

Theorem 3. $\Pi^{[\mathbb{R}]}_{\mathsf{L2\text{-}oss}}$ *securely realizes* $\mathcal{F}^{[\mathbb{R}]}_{\mathsf{oss}}$ *in the* $(\mathcal{F}_{\mathsf{Coin}}, \mathcal{F}^{\langle\cdot\rangle}, \mathcal{F}_{\mathsf{GenAST}})$*-hybrid model.*

5.3 Summary and Discussion

We stress the fundemental distinction between our technique and prior works [23, 39, 53]. Previous works apply *unauthenticated* correlations over authenticated secret sharings in a non-ABB manner, leaving them vulnerable to "inject-then-remove" selective-failure attacks by a malicious sender. In contrast, our design applies *authenticated* correlations over authenticated secret sharings, leveraging ALSS mechanisms in a black-box ABB fashion to enforce honest behavior for all *sender-side* operations and prevent online-phase selective-failure attacks. Our design only leaves a security hole on the receiver side. The only exception arises when a malicious receiver provides an inconsistent input to $\mathcal{F}^{\langle\cdot\rangle}_{\mathsf{Input}}$ when resharing back to the authenticated world. Following the observation, we propose two efficiency-preserving consistency checks. Looking back, these consistency checks cannot be obtained without our modular ABB-style design. We believe both the concept and check techniques introduce fresh air in the design space of SPDZ-family oblivious protocols, which may be of independent interest.

6 Security Analysis for the Protocol from Gao *et al.* [24]

Gao *et al.* [24] recently proposed a new CGP-like shuffle protocol with claimed malicious security and linear cost. We summarize [24] in the two-party setting, which relies on the following correlation:

$$\begin{bmatrix} \langle\beta\rangle & \langle \boldsymbol{r_0}\rangle & \langle\beta \boldsymbol{r_0}\rangle & \langle\pi_1(\boldsymbol{r}_1)\rangle \\ & & \pi_0 & \pi_1 \\ \langle \boldsymbol{r}'_0\rangle & \langle \boldsymbol{r}'_1\rangle & \langle\pi_0(\boldsymbol{r'_0})\rangle & \langle\pi_1(\boldsymbol{r'_1})\rangle \\ & & & \boldsymbol{z_1} \end{bmatrix}$$

In this correlation, π_i is a random sampled n-swap permutation held by P_i ($i \in \{0,1\}$). $\boldsymbol{z}_1 = (\boldsymbol{z}^1_1, \boldsymbol{z}^2_1)$ is held by P_1, where $\boldsymbol{z}^j_i$ ($j \in \{0,1\}$) is a vector of length n, under the following constraints:

$$\boldsymbol{z}^1_1 = \pi_0(\boldsymbol{r}_0) - \boldsymbol{r}_1, \boldsymbol{z}^2_1 = \pi_0(\beta \boldsymbol{r}_0) - \beta \boldsymbol{r}_1 + \pi_0(\boldsymbol{r}'_0) - \boldsymbol{r}'_1.$$

Gao *et al.* proposes an offline protocol to generate the above correlation, which is not our focus because our attack applies to [24] even if the correlations are securely generated (*e.g.*, distributed by a trusted dealer). Therefore, we assume the above correlations are correctly correlated in our attack. Suppose the parties want to shuffle a vector $\langle \boldsymbol{x}\rangle$.Thee parties run the following protocol:

1. $\langle \beta \boldsymbol{x}\rangle \leftarrow \langle\beta\rangle \cdot \langle \boldsymbol{x}\rangle$.

2. $\langle \boldsymbol{z}_0^1 \rangle \leftarrow \langle \boldsymbol{x} \rangle - \langle \boldsymbol{r_0} \rangle$, $\langle \boldsymbol{z}_0^2 \rangle \leftarrow \langle \beta \boldsymbol{x} \rangle - \langle \beta \boldsymbol{r_0} \rangle - \langle \boldsymbol{r}'_0 \rangle$.
3. Open $\boldsymbol{z}_0 = (\boldsymbol{z}_0^1, \boldsymbol{z}_0^2)$ to P_0. ▷ $\boldsymbol{z}_0$ is kept secret from P_1
4. P_0 computes and broadcasts $\boldsymbol{y}_0 = \pi_0(\boldsymbol{z}_0)$.
5. P_1 computes and broadcasts $\boldsymbol{y}_1 = \pi_1(\boldsymbol{z}_1 + \boldsymbol{y}_0)$.
6. Run Verify$(\boldsymbol{y}_i^1, \boldsymbol{y}_i^2, \langle \beta \rangle, \langle \pi_i(\boldsymbol{r}'_i) \rangle)$ for $i \in \{0, 1\}$.
7. The parties share $\langle \boldsymbol{y}_1^1 \rangle$ from $\boldsymbol{y}_1^1$. ▷ local computation
8. $\langle \boldsymbol{x}' \rangle \leftarrow \langle \boldsymbol{y}_1^1 \rangle + \langle \pi_1(\boldsymbol{r}_1) \rangle$.

The verification protocol is defined as follows.

Verify$(\boldsymbol{y}_i^1, \boldsymbol{y}_i^2, \langle \beta \rangle, \langle \pi_i(\boldsymbol{r}'_i) \rangle)$:

1. $\langle d \rangle \leftarrow \boldsymbol{y}_i^1 \cdot \langle \beta \rangle - \boldsymbol{y}_i^2 - \langle \pi_i(\boldsymbol{r}'_i) \rangle$
2. Open $\langle d \rangle$ for all parties.
3. Return normally if $d = 0$; abort otherwise.

If all parties behave honestly, each party P_i holds $\boldsymbol{z}_i$ such that:

$$(\boldsymbol{z}_0^1, \boldsymbol{z}_0^2) = (\boldsymbol{x} - \boldsymbol{r}_0, \beta \boldsymbol{x} - \beta \boldsymbol{r}_0 - \boldsymbol{r}'_0),$$
$$(\boldsymbol{z}_1^1, \boldsymbol{z}_1^2) = (\pi_0(\boldsymbol{r}_0) - \boldsymbol{r}_1, \pi_0(\beta \boldsymbol{r}_0) - \beta \boldsymbol{r}_1 + \pi_0(\boldsymbol{r}'_0) - \boldsymbol{r}'_1).$$

And the message $\boldsymbol{y}_i$ broadcasted by party P_0 and P_1 are

$$(\boldsymbol{y}_0^1, \boldsymbol{y}_0^2) = (\pi_0(\boldsymbol{x}) - \pi_0(\boldsymbol{r}_0), \pi_0(\beta \boldsymbol{x}) - \pi_0(\beta \boldsymbol{r}_0) - \pi_0(\boldsymbol{r}'_0)),$$
$$(\boldsymbol{y}_1^1, \boldsymbol{y}_1^2) = (\pi_1(\pi_0(\boldsymbol{x})) - \pi_1(\boldsymbol{r}_1), \pi_1(\pi_0(\beta \boldsymbol{x})) - \pi_1(\beta \boldsymbol{r}_1) - \pi_1(\boldsymbol{r}'_1)).$$

One can check the correctness $\boldsymbol{y}_1^1 + \langle \pi_1(\boldsymbol{r}_1) \rangle = \pi_1(\pi_0(\boldsymbol{x}))$.

As for privacy of the shared secrets, the intuition behind the design is that the opened values $\boldsymbol{z}_0^1 = \boldsymbol{x} - \boldsymbol{r}_0$ and $\boldsymbol{z_0^2} = \beta \boldsymbol{x} - \beta \boldsymbol{r_0} - r'_1$ reveals no information about $\boldsymbol{x}$ and $\beta \boldsymbol{x} - \beta \boldsymbol{r_0}$, as $\boldsymbol{r_0}$ and $\boldsymbol{r'_1}$ are shared between and they are hidden from any party.

The Proposed Attack. Intuitively, the protocol in [24] aims to ensure the integrity of $\boldsymbol{y}_i^1$ via $\boldsymbol{y}_i^2$, which encodes an SPDZ-like MAC that is masked by $\pi_i(r'_i)$. The intuition follows that even if the adversary injects errors before revealing $\boldsymbol{y}_i^1$, the error will be multiplied and randomized by the MAC key β such that the adversary cannot feasiably remove the introduced error to pass the following verification protocol (unless it can guess the MAC key β, which only happens with negligible probability).

However, a malicious adversary may inject an error into $\boldsymbol{y}_i^2$, and we show this can be exploited to perform selective failure attacks from the corrupted P_0, allowing P_0 to learn sensitive information about π_1 with non-negligible probability. The attack is sketched as follows:

- Instead of broadcasting $(\boldsymbol{y}_0^1, \boldsymbol{y}_0^2) = (\pi_0(\boldsymbol{x}) - \pi_0(\boldsymbol{r}_0), \pi_0(\beta \boldsymbol{x}) - \pi_0(\beta \boldsymbol{r}_0) - \pi_0(\boldsymbol{r}'_0))$, P_0 broadcasts $(\boldsymbol{y}_0^1, \boldsymbol{y}_0^2) = (\pi_0(\boldsymbol{x}) - \pi_0(\boldsymbol{r}_0), \pi_0(\beta \boldsymbol{x}) - \pi_0(\beta \boldsymbol{r}_0) - \pi_0(\boldsymbol{r}'_0) + \boldsymbol{e})$, where $\boldsymbol{e}$ is an one-hot error vector with non-zero element appearing at position p.

- The honest P_1, after receiving the errored $(\boldsymbol{y}_0^1, \boldsymbol{y}_0^2) = (\pi_0(\boldsymbol{x}) - \pi_0(\boldsymbol{r}_0), \pi_0(\beta\boldsymbol{x}) - \pi_0(\beta\boldsymbol{r}_0) - \pi_0(\boldsymbol{r}'_0) + \boldsymbol{e})$, computes $\boldsymbol{y}_1 = \pi_1(\boldsymbol{z}_1 + \boldsymbol{y}_0) = (\pi_0(\boldsymbol{r}_0) - \boldsymbol{r}_1, \pi_0(\beta\boldsymbol{r}_0) - \beta\boldsymbol{r}_1 + \pi_0(\boldsymbol{r}'_0) - \boldsymbol{r}'_1) + \pi_1(\boldsymbol{e})$.
- Before checking $\boldsymbol{y}_1$, P_0 guesses $\pi_1(\boldsymbol{e})$ by simply guessing the permuted position of the non-zero element, and remove the guessed $\pi_1(\boldsymbol{e})$ from P_0's local share of $\langle \boldsymbol{d} \rangle$. If guessed correctly, P_0 would exactly remove the error and pass the check, which happens with probability $1/n$ as $\boldsymbol{e}$ is a one-hot vector.

This attack demonstrates [24] cannot achieve full privacy. In particular, [24] attempted to authenticate online protocol messages using a tailored SPDZ-like MAC. It seems that the new tailored MAC protects the integrity of the protocol message $\boldsymbol{y}_i^1$. However, a maliciously corrupted P_0, when playing the role of a sender, may inject selective failure errors into the MAC message $\boldsymbol{y}_0^2$, and remove its permuted version from its local share *before* the final integrity check.

7 Performance Evaluation

7.1 Implementation

We implement MOSAC and [53] in C++.[3] Our implementation relies on malicious security mechanisms for SPDZ ALSS. We implement protocols for ALSS with a MASCOT-style [36] preprocessing, with SPDZ sacrifice to generate Beaver multiplication triples. Based on that, we implement necessary protocols for maliciously secure computation over ALSS, including protocols to securely realize $\mathcal{F}_{\mathsf{Input}}^{\langle\cdot\rangle}$, $\mathcal{F}_{\mathsf{Rand}}^{\langle\cdot\rangle}$, $\mathcal{F}_{\mathsf{Triple}}^{\langle\cdot\rangle}$, $\mathcal{F}_{\mathsf{LinCom}}^{\langle\cdot\rangle}$, $\mathcal{F}_{\mathsf{Mul}}^{\langle\cdot\rangle}$, $\mathcal{F}_{\mathsf{OpenCheck}}^{\langle\cdot\rangle}$. We implement both [53] and MOSAC under the same codebase in order to fairly compare two schemes. For MOSAC, we additionally employ the half-tree optimization [28] to optimize its correlation generation. We implement [53] and MOSAC with GBN configuration. Our implementation employs GBN decomposition [13] to make a trade-off between computation and communication for the correlation preprocessing.

7.2 Complexity Analysis

Computation and Communication. We summarize CGP-family shuffle protocols [13,23,39,53] and MOSAC in Table 1. We compare MOSAC with previous works [13,23,39,53] with GBN decomposition [13], where T is the dimension of the small permutations when decomposing an n-swap permutation. MOSAC enjoys improved efficiency than [53] for online communication and computation as MOSAC does not require repeated online execution.

Storage and Memory Cost Analysis. We note that MOSAC requires much less storage to store shuffle-enabled correlations than [53]. In particular, [53] can only conduct leakage-reduction in the shuffle phase to remove possible leakage. For each shuffle, the parties must generate and maintain B shuffle tuples before

[3] https://github.com/liang-xiaojian/MOSAC.

Table 1. Comparison of CGP-like shuffle protocols. n denotes the dimension of vectors; T denotes the dimension of small permutations in the GBN decomposition; B denotes the cut-and-choose bucket size. Half check $\sqrt{\times}$ indicates that [23,39] achieve correctness but do not guarantee full privacy.

	Malicious Security	Communication		Computation		Round	
		Offline	Online	Offline	Online	Offline	Online
[13]	×	$O(\kappa n \log n/\log T)$	$O(\ell n)$	$O(Tn\log n/\log T)$	$O(n)$	$O(\log n)$	$O(1)$
[39]	$\sqrt{\times}$	-	$O(\ell n)$	-	$O(n)$	$O(\log n)$	$O(1)$
[23]	$\sqrt{\times}$	$O(\ell n \log n)$	$O(\ell n)$	$O(n^2)$	$O(n)$	$O(\log n)$	$O(1)$
[53]	√	$O(B\kappa n\log n/\log T)$	$O(B\ell n)$	$O(BTn\log n/\log T)$	$O(Bn\log n/\log T)$	$O(\log n)$	$O(1)$
MOSAC	√	$O(B\kappa n\log n/\log T + n\kappa\ell)$	$O(\ell n)$	$O(BTn\log n/\log T)$	$O(n)$	$O(\log n)$	$O(1)$

using them for the online phase. In the cut-and-choose-based correlation generation protocol with batch size N, this requires the parties to store BN shuffle tuples on disk storage (if correlations are stored on dish) or in RAM memory (if correlations are stored in memory), while MOSAC only requires the parties to maintain N AST correlations after the offline AST generation (plus other correlations including Beaver triples, which is much cheaper than storing ASTs). Indeed, [53] requires more storage or memory to maintain shuffle tuples if the GBN optimization is used. In the GBN decomposition, an n-swap permutation π into $d = 2\lceil \frac{\log n}{\log T} \rceil - 1$ layers of n-swap permutation, and each layer consists of n/T T-swap permutations. Therefore, each n-swap permutation is decomposed into $(2\lceil \frac{\log n}{\log T} \rceil - 1) \cdot \frac{n}{T}$ permutations. When GBN is applied to [53], each T-swap permutations requires B T-swap permutations. This means that the parties must maintain $(2\lceil \frac{\log n}{\log T} \rceil - 1) \cdot \frac{nB}{T}$ shuffle tuples for shuffle-phase leakage reduction. In contrast, the parties only maintain N ASTs of n dimensions in MOSAC. Even for the peak memory consumption, our performance evalaution shows that the peak memory consumption for MOSAC and [53] in the expensive offline phase is about 1.6 GB and 4.4 GB, respectively, for $n = 2^{20}$ and $T = 64$.

7.3 Concrete Efficiency

Experiment Setting. We experiment on a PC equipped with Intel(R) Core(TM) i7-9750H CPU @ 2.60 GHz × 14 running Ubuntu 20.04 with 64 GB memory. We simulate a local-area network (LAN, RTT: 0.2 ms, 10 Gbps) and a wide-area network (WAN, RTT: 20 ms, 500 Mbps) by Linux `tc` command. The experiment is conducted in a two-party setting with $\kappa = 128$, $\lambda = 40$, and $\ell = 64$ (we use $\mathbb{F}_p$ for Mersenne prime $p = 2^{61} - 1$) and $\ell = 128$ (we use $\mathbb{F}_p$ for Mersenne prime $p = 2^{127} - 1$). We compare the efficiency of MOSAC to [53] with GBN configurations of (n, T). We take $n \in \{2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}\}$. We use MOSAC-1 to denote the MOSAC shuffle via permutation check and MOSAC-2 to denote the MOSAC shuffle via layer-two MAC check.

Shuffle Performance. We report the shuffle efficiency between MOSAC and [53] in Table 2. Due to online repeated shuffle, [53] requires more communication than MOSAC. In particular, [53] requires about 8× communication than

Table 2. Shuffle communication (MB) and running time (s).

		$\ell = 64$					$\ell = 128$				
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Time (LAN)	[53]	0.03	0.07	0.24	1.96	5.93	0.06	0.22	0.57	4.62	26.76
	MOSAC-1	0.02	0.06	0.22	0.68	2.52	0.04	0.15	0.45	1.66	8.06
	MOSAC-2	0.01	0.03	0.12	0.35	1.26	0.02	0.06	0.24	0.73	3.59
Time (WAN)	[53]	0.34	0.60	1.28	8.52	23.02	0.53	0.94	2.52	17.24	47.85
	MOSAC-1	0.39	0.61	1.04	2.24	6.94	0.52	0.79	1.47	4.09	16.24
	MOSAC-2	0.27	0.44	0.70	1.44	4.23	0.34	0.54	1.01	2.40	8.73
Comm.	[53]	2.06	6.75	24.00	200.00	560.00	4.13	13.50	48.00	400.00	2016.00
	MOSAC-1	0.47	1.88	7.50	30.00	120.00	0.94	3.75	15.00	60.00	240.00
	MOSAC-2	0.31	1.25	5.00	20.00	80.00	0.63	2.50	10.00	40.00	160.00

MOSAC for $n = 2^{20}$. As for running time, MOSAC is around 6× faster than [53]. We note that for small dimensions $n = 2^{12}$, MOSAC is sometimes slower than [53] in the WAN setting. The reason is that [53] entitles a one-round shuffling optimization [13,53] even it requires repeated shuffling, whereas MOSAC-1 still requires at least 17 rounds even after using communication optimization including batch processing and delay-check optimizations in our implementation; though both MOSAC and [53] enjoy asymptotic $O(1)$ round complexity. Therefore, MOSAC is much faster over large-dimension vectors. This shows that MOSAC scales better than [53] when shuffling large ALSS vectors.

When comparing MOSAC-1 with MOSAC-2, MOSAC-2 is more efficient than MOSAC-1 as the online phase of MOSAC-2 does not require n-input multiplication evaluation. This shows that even MOSAC-2 requires layer-two MAC, the overall running time and communication is still less than MOSAC-1. Overall, MOSAC-2 is around 2× faster than MOSAC-1 in running time.

Table 3. Preprocessing communication (MB) and running time (s).

		$\ell = 64$					$\ell = 128$				
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Time (LAN)	[53]	5.76	25.72	162.38	635.51	2683.86	5.76	25.72	162.38	635.51	2683.86
	MOSAC-1	8.18	19.68	84.95	347.6	1780.28	12.96	29.34	114.60	476.82	2384.21
	MOSAC-2	7.03	22.30	99.24	301.56	1672.07	9.05	26.18	103.99	340.17	1879.01
Time (WAN)	[53]	6.08	26.80	120.23	680.04	2794.41	6.27	28.01	124.64	661.13	2756.38
	MOSAC-1	78.16	73.01	200.85	869.56	3867.06	87.94	130.12	314.61	1312.69	5696.83
	MOSAC-2	60.23	120.09	265.81	828.81	3772.68	62.60	132.80	270.20	875.65	3832.65
Comm.	[53]	33.00	125.54	504.03	2256.12	11760.60	33.00	125.54	504.03	2256.12	11760.60
	MOSAC-1	99.72	399.17	1593.57	7451.45	34207.96	250.31	999.58	3987.61	17307.72	74513.18
	MOSAC-2	31.16	123.83	748.54	2993.09	11482.81	41.30	160.60	616.27	3981.52	14635.07

Preprocessing Performance. We also report the preprocessing performance. Both MOSAC-1 and MOSAC-2 require less running time than [53] in the LAN setting. The speedup is mainly due to that MOSAC employs computational optimizations, *e.g.*, using half-tree [28] for correlation generation, which requires less computational overhead than [53]. However, both MOSAC-1 and MOSAC-2 are slower than [53] in the WAN setting. In particular, MOSAC-1 requries around 2× running time than [53], while MOSAC-2 requires around 1.5× running time than [53]. The reason is that MOSAC requires more communication volume and rounds in the preprocessing. In particular, generating random ALSS shairings, Beaver's MTs, and n-input multiplication is concretely expensive in the WAN setting with higher latency. This is also reflected between MOSAC-2 and MOSAC-1: MOSAC-2 runs faster and requires less communication than MOSAC-1.

Breakdown Preprocessing Efficiency. To understand the preprocessing bottlenecks and provide insights for future optimizations, Fig. 4 reports the breakdown efficiency of main components in MOSAC-1 and MOSAC-2. We report efficiency of AST preprocessing, n-MUL preprocessing, and MT processing for MOSAC-1. For MOSAC-2, we report AST and MT preprocessing.

Table 4. Breakdown preprocessing communication (MB) and running time (s).

			$\ell = 64$					$\ell = 128$				
			2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Time (LAN)	MOSAC-1	AST	6.36	14.57	66.28	271.25	1770.97	8.38	16.02	65.36	283.63	1624.54
		n-MUL	1.41	3.53	12.39	47.60	193.29	3.31	9.27	33.38	124.70	504.77
		MT	0.39	1.59	6.25	28.72	97.35	1.22	4.03	15.83	68.49	254.85
	MOSAC-2	AST	7.01	18.06	94.92	301.56	1667.77	9.03	20.38	98.49	341.14	1873.26
		L2-MAC	4.30	4.24	4.32	4.27	4.30	5.70	5.80	5.50	5.60	5.72
Time (WAN)	MOSAC-1	AST	59.55	68.53	124.95	619.12	2752.89	62.05	70.84	126.08	615.76	2803.39
		n-MUL	17.10	26.29	56.59	174.24	752.90	21.99	44.92	132.26	473.24	1942.33
		MT	1.48	4.93	19.28	76.17	361.24	3.90	14.32	56.23	223.64	951.11
	MOSAC-2	AST	60.22	72.98	209.20	772.71	3716.70	62.56	75.26	204.15	818.28	3774.82
		L2-MAC	56.30	56.50	56.30	56.20	56.98	57.40	57.54	57.51	57.37	57.80
Comm.	MOSAC-1	AST	26.10	105.04	417.42	2746.84	11142.28	31.17	123.42	483.43	3290.99	18447.14
		n-MUL	49.11	196.13	784.14	3136.18	12545.12	146.11	594.13	2336.16	9344.23	37377.39
		MT	24.51	98.01	392.01	1568.44	6272.5	73.02	292.02	1168.03	4672.49	18688.64
	MOSAC-2	AST	31.16	123.41	484.24	2992.68	11482.40	41.30	160.17	615.85	3981.10	14634.65
		L2-MAC	0.41	0.41	0.41	0.41	0.41	0.42	0.42	0.42	0.42	0.42

We can see that n-MUL and MT preprocessing contribute substantially more overhead in MOSAC-1 than in MOSAC-2, especially over WAN. For example, at $n = 2^{20}$ and $\ell = 128$, n-MUL and MT together take 2893.44 s (about 50% of the total 5696.83s) and communicate $37377.39 + 18688.64 \approx 5.61 \times 10^4$MB (Table 4), which explains the larger WAN running time of MOSAC-1 in Table 3. By contrast, the preprocessing for layer-two MAC in MOSAC-2 is essentially

latency-bound and size-independent: it communicates only 0.41–0.42 MB with a fixed 2704 rounds across all n and ℓ, resulting in a nearly flat WAN time of $\approx$ 57–58 s (and $\approx$ 5–6 s on LAN). The reason is that the layer-two MAC only requires the multiplication between a single layer-two MAC key and an ALSS vector, which is much cheaper to realize via vector OLE. In our implementation, each batch of random OLE generation already produces sufficient numbers of random OLEs for computing the layer-two MAC. This is not the case for MOSAC-1. MOSAC-1 requires $O(n \log n)$ MTs rather than random OLEs for n-MUL proprocessing, which is more expensive.

The breakdown suggests two directions for future optimizations on preprocessing: (1) reduce the concrete round complexity for AST generation, and (2) keep WAN-facing subroutines constant-round (as in the OLE-based MT), which empirically stabilizes the WAN running time.

Summary and Discussion. We can conclude that MOSAC and [53] entail different trade-offs on various aspects. MOSAC aims to improve the shuffle efficiency while requiring more overhead for the offline correlation generation, which seems to be an inherent price to trade for an authenticated and efficient online shuffle protocol. Achieving an efficient online phase is a common practice for secret-sharing-based MPC [19–21,36], by moving relatively expensive correlation preprocessing to the offline phase before performing actual computation, which is also a design principle from CGP shuffle. Such a trade-off is particularly appealing for SSS-based applications. For instance, Camel [55] relies on SSS for secure aggregation to enhance privacy in federated learning (FL). In FL, users perform local model training before aggregating local models into a global model in a round-by-round fashion. Faster online shuffle-based aggregation is critical for minimizing costs and delays. In contrast, the time interval between rounds is typically long to accommodate time-consuming local training, between which the shuffling servers can preprocess input-independent correlations. Another example is anonymous communication systems. In Clarion [23], shuffling servers execute shuffling after gathering enough messages to create a large anonymous set. The servers can precompute offline correlations while waiting for messages, and a faster online shuffle means faster message delivery. Last but not least, MOSAC's online-efficient design may benefit applications designed under the two-server model with distributed trust [12,17,51,55], where (not necessarily trusted) dealers can distribute correlations cheaply, without needing correlation preprocessing between servers.

8 Related Work

Shuffle is perhaps mostly related to the seminal work on mix-net from Chaum [14] in the early age of modern cryptography. In such systems, the senders encrypt messages using the public keys of shuffling servers sequentially, and the shuffling servers decrypt and shuffle the messages in sequence. Correct shuffling can be maintained by zero-knowledge proof (ZKP).

Secret-shared shuffle focuses on shuffling secret-shared data. SSS can be realized by securely evaluating a permutation network [10,54] from existing works [45–47]. This approach incurs communication complexity $O(\ell n \log n)$ and round complexity $O(\log n)$ for secret-sharing-based MPC. It's possible to design SSS from additive homomorphic encryption (AHE) [32,44]. The parties convert the secret-shared vector to AHE ciphertexts, permute the ciphertexts, and convert the permuted ciphertexts back to the secret-sharing form. This approach achieves asymptotically linear communication. However, public-key operations incur higher computational overhead. In the context of malicious security [44], each party proves correct shuffling via ZKP, which is computationally expensive.

CGP shuffle [13] presents a nice balance between communication and computation. CGP shuffle mostly relies on symmetric primitives (*e.g.*, PRG) and OT to generate shuffle correlations, which is much computationally cheap than the AHE-based computation. For communication, it only incurs linear and one-round online communication. Besides, CGP shuffle can be configured with GBN decomposition to achieve flexible trade-offs between communication and computation. Many recent works [6,23,33,38,39,41,43,50,53,56] follow the CGP paradigm to design new protocols and/or applications. Among them, two works [23,39] enhanced CGP shuffle with malicious security, but they suffer from similar attacks as studied by Song *et al.* [53]. Song *et al.* [53] relies on online leakage reduction to obtain full privacy, breaking CGP's linear communication complexity with increased online overhead.

9 Conclusion

This paper proposes a new framework for designing maliciously secure two-party secret-shared shuffle protocols with linear cost and constant-round communication in the online-offline paradigm. Our protocol achieves the claimed property by relying on a new authenticated correlation. Based on that, we propose a new shuffle paradigm and new efficiency-preserving consistency checks that eliminate online-phase selective failure attacks in existing solutions, avoiding repeated online execution. We implement our protocol and compare it with state-of-the-art work. The performance evaluation shows that our protocol is more efficient while introducing more but acceptable offline overhead.

Future Work. Our work initiates maliciously secure SSS protocols based on authenticated correlation and leaves possible directions for future research. MOSAC enjoys nice online efficiency and security properties, but with more offline overhead for correlation pre-processing. It remains to be designed more efficient pre-processing protocols. In particular, using HE-based SPDZ preprocessing may result in more efficient offline preprocessing in the WAN setting from its concretely reduced rounds. Another direction is to apply MOSAC in the dealer-available model [12,17,51] or honest majority settings, which may enable MOSAC variants with better efficiency.

Acknowledgment. We thank the anonymous reviewers for insightful suggestions and comments. This work is supported by the National Natural Science Foundation of China (Grant 62302118), the National Research Foundation, Singapore, Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme, and CyberSG R&D Cyber Research Programme Office. Any opinions, findings and conclusions or recommendations expressed in these materials are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, Cyber Security Agency of Singapore as well as CyberSG R&D Programme Office, Singapore.

References

1. Alexopoulos, N., Kiayias, A., Talviste, R., Zacharias, T.: MCMix: anonymous messaging via secure multiparty computation. In: USENIX Security Symposium, pp. 1217–1234 (2017)
2. Anderson, E., Chase, M., Durak, F.B., Laine, K., Weng, C.: Precio: private aggregate measurement via oblivious shuffling. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, pp. 1819–1833 (2024)
3. Araki, T., Furukawa, J., Ohara, K., Pinkas, B., Rosemarin, H., Tsuchida, H.: Secure graph analysis at scale. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 610–629 (2021)
4. Asharov, G., et al.: Efficient secure three-party sorting with applications to data analysis and heavy hitters. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp. 125–138 (2022)
5. Asharov, G., Hamada, K., Kikuchi, R., Nof, A., Pinkas, B., Tomida, J.: Secure statistical analysis on multiple datasets: join and group-by. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, pp. 3298–3312 (2023)
6. Attrapadung, N., et al.: Oblivious linear group actions and applications. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 630–650 (2021)
7. Balle, B., Bell, J., Gascón, A., Nissim, K.: Private summation in the multi-message shuffle model. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020, pp. 657–676 (2020)
8. Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 263–280. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29011-4_17
9. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 420–432. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_34
10. Beneš, V.E.: Optimal rearrangeable multistage connecting networks. Bell Syst. Tech. J. **43**(4), 1641–1656 (1964)
11. Bogdanov, D., Laur, S., Talviste, R.: A practical analysis of oblivious sorting algorithms for secure multi-party computation. In: Bernsmed, K., Fischer-Hübner, S. (eds.) NordSec 2014. LNCS, vol. 8788, pp. 59–74. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11599-3_4

12. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Lightweight techniques for private heavy hitters. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 762–776. IEEE (2021)
13. Chase, M., Ghosh, E., Poburinnaya, O.: Secret-shared shuffle. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020. LNCS, vol. 12493, pp. 342–372. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64840-4_12
14. Chaum, D.: Untraceable electronic mail, return addresses, and digital pseudonyms. Commun. ACM **24**(2), 84–88 (1981)
15. Cheu, A., Smith, A., Ullman, J., Zeber, D., Zhilyaev, M.: Distributed differential privacy via shuffling. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11476, pp. 375–403. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17653-2_13
16. Cheu, A., Zhilyaev, M.: Differentially private histograms in the shuffle model from fake users. In: IEEE SP, pp. 440–457 (2022)
17. Corrigan-Gibbs, H., Boneh, D.: Prio: private, robust, and scalable computation of aggregate statistics. In: NSDI, pp. 259–282 (2017)
18. Dalskov, A., Orlandi, C., Keller, M., Shrishak, K., Shulman, H.: Securing DNSSEC keys via threshold ECDSA from generic MPC. In: Chen, L., Li, N., Liang, K., Schneider, S. (eds.) ESORICS 2020. LNCS, vol. 12309, pp. 654–673. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59013-0_32
19. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority – or: breaking the SPDZ limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 1–18. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40203-6_1
20. Damgård, I., Orlandi, C.: Multiparty computation for dishonest majority: from passive to active security at low cost. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 558–576. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_30
21. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_38
22. Erlingsson, Ú., Feldman, V., Mironov, I., Raghunathan, A., Talwar, K., Thakurta, A.: Amplification by shuffling: from local to central differential privacy via anonymity. In: Chan, T.M. (ed.) SODA, pp. 2468–2479 (2019)
23. Eskandarian, S., Boneh, D.: Clarion: anonymous communication from multiparty shuffling protocols. In: Network and Distributed System Security (NDSS) Symposium (2022)
24. Gao, J., Zhang, Y., Zhong, S.: Multiparty shuffle: linear online phase is almost for free. Cryptology ePrint Archive (2024)
25. Garimella, G., Mohassel, P., Rosulek, M., Sadeghian, S., Singh, J.: Private set operations from oblivious switching. In: Garay, J.A. (ed.) PKC 2021. LNCS, vol. 12711, pp. 591–617. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-75248-4_21
26. Gascón, A., Ishai, Y., Kelkar, M., Li, B., Ma, Y., Raykova, M.: Computationally secure aggregation and private information retrieval in the shuffle model. Cryptology ePrint Archive (2024)
27. Genkin, D., Ishai, Y., Prabhakaran, M.M., Sahai, A., Tromer, E.: Circuits resilient to additive attacks with applications to secure computation. In: Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing, pp. 495–504 (2014)

28. Guo, X., et al.: Half-tree: halving the cost of tree expansion in cot and DPF. In Hazay, C., Stam, M. (eds) EUROCRYPT 2023. LNCS, vol. 14004, pp. 330–362. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30545-0_12
29. Hamada, K., Ikarashi, D., Chida, K., Takahashi, K.: Oblivious radix sort: an efficient sorting algorithm for practical secure multi-party computation. Cryptology ePrint Archive (2014)
30. Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: Practically efficient multi-party sorting protocols from comparison sort algorithms. In: Kwon, T., Lee, M.-K., Kwon, D. (eds.) ICISC 2012. LNCS, vol. 7839, pp. 202–216. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37682-5_15
31. Han, F., et al.: Concretely efficient correlated oblivious permutation. Cryptology ePrint Archive (2025)
32. Huang, Y., Evans, D., Katz, J.: Private set intersection: are garbled circuits better than custom protocols? In: NDSS (2012)
33. Jia, Y., Sun, S., Zhou, H.-S., Du, J., Gu, D.: Shuffle-based private set union: faster and more secure. In USENIX Security 2022, pp. 2947–2964 (2022)
34. Keller, M.: MP-SPDZ: a versatile framework for multi-party computation. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 1575–1590 (2020)
35. Keller, M., Orsini, E., Scholl, P.: Actively secure OT extension with optimal overhead. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9215, pp. 724–741. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47989-6_35
36. Keller, M., Orsini, E., Scholl, P.: MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 830–842 (2016)
37. Koti, N., Kukkala, V.B., Patra, A., Gopal, B.R., Sangal, S., et al.: Ruffle: rapid 3-party shuffle protocols. In: Proceedings on Privacy Enhancing Technologies (2023)
38. Laud, P.: Efficient permutation protocol for MPC in the head. In: Roman, R., Zhou, J. (eds.) STM 2021. LNCS, vol. 13075, pp. 62–80. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-91859-0_4
39. Laud, P.: Linear-time oblivious permutations for SPDZ. In: Conti, M., Stevens, M., Krenn, S. (eds.) CANS 2021. LNCS, vol. 13099, pp. 245–252. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92548-2_13
40. Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In: Lai, X., Zhou, J., Li, H. (eds.) ISC 2011. LNCS, vol. 7001, pp. 262–277. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24861-0_18
41. Le, T., Hoang, T.: MAPLE: a metadata-hiding policy-controllable encrypted search platform with minimal trust. Cryptology ePrint Archive (2023)
42. Lindell, Y.: Fast cut-and-choose-based protocols for malicious and covert adversaries. J. Cryptol. **29**(2), 456–490 (2016)
43. Lu, D., Kate, A.: RPM: robust anonymity at scale. In: Proceedings on Privacy Enhancing Technologies (2023)
44. Miao, P., Patel, S., Raykova, M., Seth, K., Yung, M.: Two-sided malicious security for private intersection-sum with cardinality. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12172, pp. 3–33. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56877-1_1
45. Mohassel, P., Rindal, P., Rosulek, M.: Fast database joins and psi for secret shared data. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 1271–1287 (2020)

46. Mohassel, P., Sadeghian, S.: How to hide circuits in MPC an efficient framework for private function evaluation. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 557–574. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_33
47. Mohassel, P., Sadeghian, S., Smart, N.P.: Actively secure private function evaluation. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8874, pp. 486–505. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45608-8_26
48. Nayak, K., Wang, X.S., Ioannidis, S., Weinsberg, U., Taft, N., Shi, E.: GraphSC: parallel secure computation made easy. In: 2015 IEEE Symposium on Security and Privacy, pp. 377–394. IEEE (2015)
49. Neff, C.A.: A verifiable secret shuffle and its application to e-voting. In: Proceedings of the 8th ACM conference on Computer and Communications Security, pp. 116–125 (2001)
50. Peceny, S., Raghuraman, S., Rindal, P., Shah, H.: Efficient permutation correlations and batched random access for two-party computation. Cryptology ePrint Archive (2024)
51. Rathee, M., Shen, C., Wagh, S., Popa, R.A.: ELSA: secure aggregation for federated learning with malicious actors. In: 2023 IEEE Symposium on Security and Privacy (SP), pp. 1961–1979. IEEE (2023)
52. Schwartz, J.T.: Fast probabilistic algorithms for verification of polynomial identities. J. ACM (JACM) **27**(4), 701–717 (1980)
53. Song, X., Yin, D., Bai, J., Dong, C., Chang, E.-C.: Secret-shared shuffle with malicious security. In: Network and Distributed System Security (NDSS) Symposium (2024)
54. Waksman, A.: A permutation network. J. ACM (JACM) **15**(1), 159–163 (1968)
55. Xu, S., Zheng, Y., Hua, Z.: Camel: communication-efficient and maliciously secure federated learning in the shuffle model of differential privacy. arXiv preprint arXiv:2410.03407 (2024)
56. Yang, Y., et al.: Maliciously secure circuit-psi via SPDZ-compatible oblivious PRF. Cryptology ePrint Archive (2024)
57. Zippel, R.: Probabilistic algorithms for sparse polynomials. In: Ng, E.W. (ed.) Symbolic and Algebraic Computation. LNCS, vol. 72, pp. 216–226. Springer, Heidelberg (1979). https://doi.org/10.1007/3-540-09519-5_73
58. Zou, Z., Liu, Z., Shan,J., Li, Q., Xu, K., Xu, M.: CoGNN: towards secure and efficient collaborative graph learning. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, pp. 4032–4046 (2024)